Sistemas Informáticos
Curso 2002-2003

Librería de Efectos 2D/3D y técnicas de renderizado bajo OpenGL

Autores:
Ángel Amigo de la Huerga
Óscar San José Alonso
Óscar Sangenís Garola

Dirigido por:
Purificación Arenas Sánchez

Facultad de Informática
Universidad Complutense de Madrid

Resumen

En este trabajo se presenta una librería gráfica implementada bajo OpenGL que incluye distintos efectos especiales 2D y 3D, encapsulando además todo el proceso de texturado de OpenGL. Entre los efectos que se incluyen tenemos el billboarding, sistemas de partículas, soporte para sprites, superficies que simulan ondas y diseño de líneas con patrones avanzados. Además la librería permite importar modelos creados con programas externos (como 3D Studio) y obtener la geometría de un modelo en un fichero header de C++. El objetivo principal de esta librería es facilitar al programador una herramienta que le permita programar aplicaciones gráficos que usen efectos especiales 2D y 3D (como video juegos) de una forma clara y eficiente.

Abstract

This work presents a graphical library implemented under OpenGL, which supports several 2D/3D special effects. Moreover, the library also encapsulates all the OpenGL texture process. Some of the implemented effects are the following: billboarding, particle systems, sprite support, wave simulating surfaces and line drawing with advanced patterns. This library allows to import models generated by third party programs (such as 3D Studio) and to get the geometry of a 3D model in a C++ header file. The main aim of this library is to facilitate the design of graphics applications using 2D/3D special effects, such as video games, improving also the efficiency and clarity of programs.

Palabras clave

Informática gráfica, efectos 2D y 3D, OpenGL, texturas, iluminación.

Índice

Prefacio

El escenario tecnológico actual, tanto tecnológica como socialmente, ha propiciado una expansión del interés por diversas aplicaciones de la programación gráfica, tales como pueden ser los videojuegos, modeladores, sistemas de postproducción gráfica, etc. Las posibilidades de este tipo de aplicaciones son muchas ya que impactan directamente en el usuario a través de la vista.

Pero debido así mismo a esta expansión podemos encontrarnos una situación de alta competitividad, que demanda así mismo una alta productividad. Se necesitan productos de alta calidad con un coste tanto en tiempo como en recursos lo más bajo posible.
Para ello los programadores utilizan interfaces de desarrollo como OpenGL, que proporciona una forma simplificada y abstracta de generación de aplicaciones gráficas.
Aun así, observamos que en muchos procesos de desarrollo una parte de las tareas tiende a repetirse, ocasionando una pérdida de productividad ya que se debe emplear el tiempo de los programadores en implementar rutinas y casos de uso que son muy comunes.

Por ello surge la necesidad de crear un nivel más alto de abstracción entre el interfaz proporcionado por OpenGL (o cualquier API de desarrollo gráfico) y el usuario, enfocado hacia cierto tipo de aplicaciones gráficas que se encuentran entre las que más recursos demandan: los videojuegos. En este sector se debe emplear el tiempo y potencia de computación en innovar, ya que la industria no puede permitirse malgastar recursos en repetir diseños e implementaciones ya realizadas, bien por la propia compañía, bien por terceros.

Pero este librería debe ser lo suficientemente flexible dentro de su campo como para ser atractiva al uso por parte de los programadores. Deben por tanto identificarse los puntos que pueden conseguir un mayor interés, bien sea por lo común de su uso o por la dificultad de su implementación.

El objetivo de este proyecto se basa en estas necesidades. Pretendemos identificar los problemas y situaciones típicas a los que un programador de aplicaciones gráficas debe típicamente enfrentarse, y debemos abstraerlos de tal forma que incluyan y se adapten sino a todas, al menos a una gran mayoría de estas situaciones. Debemos así mismo proporcionar una forma de uso intuitiva y potente, que no requiera de conocimiento avanzado sobre el funcionamiento del efecto, utilidad u objeto en concreto. De este modo pretendemos crear una librería que resulte de utilidad a una gran variedad de perfiles de desarrollador, tanto el avanzado (que se vería atraído por el hecho de tener un efecto ampliamente utilizado dispuesto para su uso de una forma sencilla y con un resultado eficiente) como al amateur (que buscaría más el impacto visual y la posibilidad de utilizar efectos avanzados sin esfuerzo y requiriendo un conocimiento mínimo sobre su uso).

Para ello hemos dividido el trabajo en una serie de módulos que proporcionan cada uno una funcionalidad en concreto. Hemos pretendido que la interdependencia entre ellos

sea mínima, de tal modo que quien quiera utilizarlo únicamente necesite conocer e incluir uno de ellos en su proyecto. Sin embargo, hemos tenido en cuenta que ciertos tipos de funcionalidades son complementarias y su uso suele presentarse junto, por ello se ha pretendido que su interacción sea lo más sencilla y fluida posible.

Los puntos que hemos aislado y considerado de mayor interés han sido numerosos, pero hemos tenido que centrarnos en una serie de ellos porque intentar abordarlos todos quedaría fuera de las pretensiones del proyecto.

Por otro lado pensamos que las funcionalidades que hemos decidido implementar presentan un interés suficiente y una utilidad posible que pueden hacer que su uso proporcione a los desarrolladores un ahorro de tiempo y recursos que lo hagan atractivo. Los técnicas tratadas en el trabajo incluyen las siguientes:

El objeto CTextura

Todas las técnicas necesitan mostrar al final imágenes. Incluimos esta clase en el proyecto porque facilitó en gran medida el desarrollo del resto de las técnicas. Gracias a la encapsulación de esta textura, nos hemos podido olvidar totalmente de factores como formato de imagen, tamaño, tipo de imagen... Además esta clase permite el acceso con pocas llamadas a extensiones de OpenGL.

BillBoarding y Sprites

Una vez desarrollada la parte de apoyo del texturado, comenzamos con el estudio de efectos bidimensionales. En un primer momento se puede pensar que este tipo de aplicaciones no presenta interés por que su uso parece obsoleto, pero no es así. Una gran cantidad de aplicaciones gráficas utilizan diversos efectos bidimensionales debido a su sencillez, y su buena relación realismo/eficiencia, sobre todo en situaciones concretas. Estas situaciones suelen requerir 'engañar' al espectador presentando el objeto bidimensional de tal forma que parezca integrado en la escena tridimensional. Para ello usamos efectos como el *billboarding*, que controlando la orientación y modo de dibujado de estos objetos planos esconde su naturaleza. Sobre esto se sustentará gran parte de la justificación del uso de gráficos bidimensionales en aplicaciones 3D.

Dentro de los usos de objetos bidimensionales se pueden identificar así mismo dos tipos muy extendidos: los 'sprites' y los 'scrolls', ambos ampliamente utilizados en la historia del desarrollo gráfico en general y del videojuego en particular. Decidimos incluirlos debido a la potencia que pueden proporcionar a la hora de añadir nivel de detalle de una forma eficiente y sencilla a una escena.

Mencionar también que la industria del videojuego se ha basado hasta hace poco en técnicas exclusivamente 2D y que mejor que poder utilizar la experiencia con la enorme potencia del hardware gráfico actual.

Mesh Loading y Modelos Compilados

Una aplicación gráfica 3D necesita de modelos en 3D por definición. Al ser OpenGL exclusivamente una librería gráfica, no ofrece soporte para cargar desde archivo modelos. Este apartado lo hemos querido solucionar con un cargador de objetos OBJ, un estándar reconocido. La carga de trabajo que se libera usando este cargador es enorme.
Por otro lado, al investigar sobre formatos de modelos 3D, observamos que se reducían a listas de números, la misma representación que usaríamos si guardamos la geometría del modelo en el código fuente. Nos asalto la idea de generar un convertidor especial, que transforma archivos de modelo 3D en archivos .h (*headers*). Este enfoque nos parece muy interesante por varias razones:

- Accedíamos a la información geométrica del modelo directamente.
- El archivo ocupaba mucho menos, ya que en el código se guarda la geometría en formato numérico mientras que en el archivo se guarda en ASCII.
- Los modelos no son accesibles desde fuera de la aplicación.
- Son mucho más rápido de cargar, ya que se realiza cuando se mueve el código binario del archivo a memoria para ejecutarlo.

Utilizamos el programa *awk*, típico de UNIX para esta conversión, algo que jamás se relacionaría con una aplicación gráfica.

Sistemas de Partículas

La técnica más utilizada con diferencia para renderizar efectos especiales es utilizar sistemas de partículas. Un sistema de partículas consiste en una colección de elementos (partículas) que modelan un fenómeno físico. Las partículas se rigen por leyes muy sencillas, pero en conjunto simulan un comportamiento mucho más complejo. Esta técnica produce efectos sorprendentes resultados, hasta tal punto que resulta complicado a primera vista pensar que están renderizados a base de polígonos. Es una herramienta indispensable para generar efectos especiales tales como fuego, humo y explosiones.

En este apartado se presenta una librería de clases, que por su diseño, permite crear nuevos efectos con extraordinaria facilidad.

Sombras

Este módulo muestra las diferentes técnicas existentes para generar sombras en OpenGL. Las sombras resultan primordiales en una aplicación que produzca escenas de apariencia real. Lógicamente, OpenGL no ofrece soporte directo para la generación de sombras. Ofrecemos en este apartado las matemáticas involucradas en el proceso y la utilización del buffer de estarcido.

Iluminación Estática

En este módulo se pretenden ilustrar las principales técnicas de iluminación estática. Cuando se hace uso de la iluminación estática en una aplicación, se obtiene una ganancia media del 50% sin sacrificar los resultados gráficos. Las técnicas tratadas son preiluminación, *lightmapping* y volúmenes de luz.

Renderizado de líneas con patrones avanzados

A veces las líneas de OpenGL y sus patrones se quedan cortos. Este apartado pretende ofrecer una técnica para poder renderizar líneas con el aspecto que el diseñador elija. Por ejemplo, un tubo de neón o un rayo podrían dibujarse como una línea que tuviera una degradación de color desde el centro de la línea hasta los costados. Todo esto acelerado por el hardware gráfico de texturado, el más importante de la tarjeta.

Simulación de efectos ondulantes

Por último, decidimos introducirnos en unas matemáticas más complejas para simular objetos deformables. Con este apartado pretendemos unir simulación física y efectos gráficos. Una bandera ondeante o una superficie de agua son objetos que se presentan en el día a día de los videojuegos y que les otorgan realismo a bajo precio.

La clase CTextura

Encapsulación del soporte de Texturizado de OpenGL

Introducción

OpenGL ofrece un amplio soporte de texturas con numerosas opciones sobre como aplicar la textura. La contrapartida es que *mapear* una simple textura necesita varias llamadas a funciones. Con la creación de una clase que encapsula la funcionalidad de las texturas, se pretende simplificar este proceso. Aparte de las ventajas que ofrece un desarrollo orientado a objetos, el programador no tendrá que recordar todos los nombres de funciones (entre 3 y 10) involucrados en el proceso de texturizado. La parte más complicada del texturizado seguramente sea cargar la imagen desde disco. Existen numerosos formatos de gráficos y cualquiera de ellos puede ser necesitado. Además si se quiere un uso eficiente del hardware gráfico se dispara el número de parámetros a tener en cuenta. Por ejemplo, el *multitexturado* requiere usar extensiones de OpenGL y el programador debe preocuparse de obtener en tiempo de ejecución las funciones necesarias. Con la encapsulación se pretende abstraer al programador de estos detalles básicos. Gracias al uso del objeto `CTextura` que ahora se detallará, la aplicación de una textura requerirá el uso de una sola función.

El Proceso de Texturado de OpenGL

Cuando se va aplicar una textura, hay que realizar los siguientes pasos:

- Cargar los bits de una imagen guardada en disco.
- Obtener un identificador único de OpenGL para referirse a la textura.
- Mover la imagen a memoria de video.
- Activar el texturado con `glEnable(GL_TEXTURE_2D).`
- Activar el identificador obtenido previamente.
- Definir los parámetros de texturado.
- Referenciar los vértices del polígono y las coordenadas de textura asociadas con dichos vértices.
- Desactivar el identificador de la textura.
- Desactivar el texturado con `glDisable(GL_TEXTURE_2D).`

Si se utiliza la clase `CTextura`, los pasos a realizar serían:

- Crear el objeto.
- Definir los parámetros de la textura.
- Utilizar un método `cargar*(…)` para cargar una imagen.
- Llamar al método activar de la textura.

- Generar la geometría del objeto (vértices y coordenadas de textura).
- Llamar al método desactivar de la textura.

Características del objeto CTextura

A continuación se muestran las características que ofrece la clase. Cada sección se refiere a un aspecto de las texturas y ofrece una descripción de los métodos involucrados.

Carga de Imágenes desde Archivo

Seguramente sea la funcionalidad más apreciada de este objeto. Cualquier aplicación con una apariencia aceptable, necesita cargar ficheros de imágenes. OpenGL lógicamente no ofrece ningún soporte y todos los programadores gráficos se habrán cruzado con este problema alguna vez. El formato de imagen más sencillo de leer es el .BMP. La clase `CTextura` ofrece soporte para los archivos GIF, JPEG, TGA y BMP. La clase tiene numerosas funciones para cargar estas imágenes y combinarlas, sobre todo en lo que respecta al canal alpha. Sólo los archivos TGA y BMP soportan el canal alpha en el archivo.

Por otro lado, la clase `CTextura` es capaz de manejar imágenes con dimensiones que no sean tamaño de potencia de 2, a diferencia de OpenGL. Cuando el objeto CTextura detecte que las dimensiones no son adecuadas, la transformará a la potencia de 2 más cercana para reducir la distorsión. Aunque pueda cargar imágenes de cualquier tamaño no significa que el hardware gráfico si pueda, por lo que al final siempre es necesaria esta transformación y se recomienda utilizar imágenes con las dimensiones adecuadas para reducir distorsión y tiempo de carga de la imagen.

Todas las funciones del tipo `cargar*(...)` eliminan cualquier textura previamente creada en ese objeto.

Función	Significado
`bool cargarImagen(char* ruta)`	Define una textura con los píxeles de la imagen almacenada en disco en la ruta *ruta*. La imagen debe tener un formato soportado por el objeto CTextura, como BMP, TGA, GIF o JPG. La imagen se guarda en memoria en el mismo formato que tiene en disco.
`bool cargarImagenAlpha(char* rutaImagen, char* rutaAlpha)`	Define una textura con los píxeles de color de la imagen almacenada en *rutaImagen*. El canal alpha se rellena con los pixels de la imagen *rutaAlpha*. La imagen rutaImagen puede tener cualquier formato de imagen, pero si tiene canal

	alpha, este sera desechado. La imagen *rutaAlpha*, lógicamente tiene que ser en escala de grises (8bpp) ya que esta destinada al canal alpha.
`bool cargarImagenAlpha(char* rutaImagen, GLuint alpha_cte)`	Define una textura con los píxeles de color de la imagen almacenados en *rutaImagen*. La componente alpha de todos los pixels se rellena con el valor especificado *alpha_cte*. Si *rutaImagen* ya contenía canal alpha, este será desechado.
`bool cargarImagenTransparente(char* ruta, GLuint rojo, GLuint verde, GLuint azul)`	Define una textura con los píxeles de color de la imagen almacenada en *rutaImagen*. La componente alpha de los píxeles que tengan como píxeles de color (rojo, verde, azul) se rellena a 0. Los demás píxeles tendrán la componente alpha a 1. Al utilizar este método, la función de mezcla se fija a `MEZCLA_PONDERADA`, ya que se supone que se quieren eliminar esas zonas de color.
`bool cargarImagenVacia(LONG anchura, LONG altura, GLuint rojo, GLuint verde, GLuint azul)`	Define una textura con las medidas especificadas *anchura***altura* y con los píxeles de color a (*rojo*, *verde*, *azul*). No posee canal Alpha.
`bool cargarImagenVaciaAlpha(LONG anchura, LONG altura, GLuint rojo, GLuint verde, GLuint azul, GLuint alpha)`	Define una textura con las medidas especificadas *anchura***altura* y con los píxeles a (*rojo*, *verde*, *azul*, *alpha*).

Carga de Imágenes desde el Framebuffer

A veces resulta útil generar una escena en OpenGL y utilizarla como textura. Por ejemplo, si se quiere realizar el efecto de "*environment mapping*" en tiempo real, se tendrá que renderizar la escena que se reflejará en el objeto reflectante y aplicarla como textura al objeto. También se suele utilizar para renderizar una sola vez un modelo que se tiene que dibujar varias veces, por ejemplo para realizar un efecto de "*motion blur*".

Función	Significado
`bool cargarImagenDesdeFrameBuffer(GLint x, GLint y, GLsizei anchura, GLsizei altura)`	Define una nueva textura con los píxeles del framebuffer que están en el recuadro con esquina inferior en la posición (*x*, *y*) y con esquina

	superior en (*x* + *anchura*, *y* + *altura*). Desecha el canal alpha del framebuffer.
`bool cargarImagenAlphaDesdeFrameBuffer( GLint` *`x`*`, GLint` *`y`*`, GLsizei` *`anchura`*`, GLsizei` *`altura`*`)`	Define una nueva textura con los píxeles del framebuffer que están en el recuadro con esquina inferior en la posición (*x*, *y*) y con esquina superior en (*x* + *anchura*, y + *altura*). NO desecha el canal alpha del framebuffer.
`copiarImagenDesdeFrameBuffer( int` *`x`*`, int` *`y`*`, int` *`anchura`*`, int` *`altura`*`)`	Copia los píxeles del framebuffer que están el recuadro con esquina inferior en la posición (*x*, *y*) y con esquina superior en (*x* + *anchura*, *y* + *altura*). La textura tiene que estar creada con anterioridad. Éste método es más rápido que los anteriores justamente porque no tiene que crear la textura.

MipMapping

El *mipmapping* es una técnica muy conocida que evita la distorsión de la imagen al alejarla o acercarla demasiado al observador. Consiste en generar una imagen con la mitad de tamaño que la anterior hasta llegar a tamaño 1x1. Se parte de la imagen inicial y mediante un filtro se generan estas imágenes llamadas *mipmaps* . Las imágenes tratadas con este filtro aparecen mucho más suaves. No es eficiente la aplicación de este filtro en tiempo real y por eso los mapas deben ser generados al especificar la textura. La función `setMipMaps` es la que activa la generación del *mipmapping*. Debe ser llamada antes que los métodos que cargan una imagen, para que en el proceso se creen los *mipmaps*.

Función	Significado
`void setMipMaps(bool` *`mipmapping`*`)`	Activa o Desactiva la generación de *mipmapping* al crear cualquier textura.

Multitexturado

El Multitexturado es una extensión a OpenGL que permite aplicar a un mismo polígono varias texturas a la vez. Para poder aplicar varias a la vez es necesario el concepto de *unidad de texturado*. Cada *unidad de texturado* se ocupa de tratar una textura. Al final del proceso se mezclan los resultados de cada unidad. El número de texturas que se pueden aplicar a la vez en un polígono es el número de *unidades de texturado* del

hardware gráfico. El hardware gráfico que soporte multitexturado debe tener al menos 2 *unidades de texturado*.

La clase Ctextura ofrece multitexturado cuando está disponible por hardware. Se puede obtener el grado de multitexturado con la función `getNumUnidadesTexturado`. Para aplicarlo, se debe asociar cada textura involucrada a una unidad de texturado distinta. Una textura se asigna a una unidad de texturado con el método `setUnidadTexturado`.

Una vez que se han asignado las unidades, hay que especificar las coordenadas de textura para cada unidad de texturado. El estándar *ARB_multitexture* define que hay que utilizar una función distinta para cada unidad, por lo que puede resultar tedioso saber en cada momento que función llamar. El método `coordTextura(...)` evita estos problemas, las coordenadas que se especifiquen serán traducidas a la unidad de texturado en la que este activada la textura. Este método también funciona con texturas normales, simplemente llamando al método `glTexCoord*.`

Función	**Significado**
`void coordTextura(float s, float t)`	Especifica las coordenadas de textura *s*, *t* de OpenGL para la unidad de texturado a la que este asignada la textura. Si no esta disponible el multitexturado, siempre se activa la unidad de texturado cero.
`int getNumUnidadesTexturado()`	Obtiene el número de unidades de texturado que posee el hardware. Como mínimo siempre devuelve 1. En este caso no estaría disponible el multitexturado por hardware.
`bool setUnidadTexturado(int unidadTex)`	Asigna a la textura a una unidad de texturado *unidadTex*. Las unidades de texturado se identifican por números desde al 0 hasta el número de unidades de texturado menos 1. Se puede obtener el número de unidades de texturado con el método anterior.

Efectos de Blending

Aunque en OpenGL los efectos de *blending* estén totalmente separados de la parte de texturas, hemos constatado que la utilización de un formato de textura suele determinar la mezcla que se aplicará al renderizar el polígono. Esta mezcla se refiere a la mezcla de los píxeles del polígono con los píxeles que ya están en el framebuffer. Por esta razón, cuando se activa el objeto textura para aplicarlo a un polígono, también se activa la función de mezcla asociada a esa textura. También se ha observado que 3 tipos de

mezclas predominan sobre cualquier otro tipo. Se han definido 3 constantes para referirse a estos tipos de mezcla, de tal forma que no sea necesario recordar los parámetros que se utilizan en la función `glBlend(…)`.

Función	Significado
`bool setMezcla(GLint tipoMezcla)`	Activa el tipo de mezcla identificado por *tipoMezcla*. Ver tabla siguiente.

Identificadores	Significado
`MEZCLA_ADITIVA`	Utilizada para sumar los píxeles de la textura a los que había previamente en el buffer. La suma se hace ponderada con el canal alpha. La función que se aplica es la especificada en OpenGL como `glBlendFunc(GL_SRC_ALPHA, GL_ONE)`
`MEZCLA_PONDERADA`	Utilizado para eliminar zonas totalmente transparente de una textura, concretamente los pixels que tengan el canal alpha a cero. También se puede utilizar para especificar la transparencia de los pixels si se especifica como componente alpha de la textura un valor mayor que cero. La función que se aplica es la especificada en OpenGL como `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`
`MEZCLA_CERO`	No aplica ningún efecto de *blending*.

Funciones de Magnification/Minification

Estas funciones se aplican cuando es necesario decidir que píxel de la textura aplicar al polígono texturado. Por ejemplo, si la textura se aplica a un polígono dos veces más grande que la textura, habrá que utilizar una interpolación para obtener los píxeles. Si se aplica a un polígono más pequeño, hay que descartar píxeles de la textura. Esta función afecta directamente a la calidad visual y al rendimiento del texturado. En la tabla se muestra por orden de rendimiento decreciente y por calidad creciente. Aunque OpenGL ofrece más posibilidades para estos filtros, sólo tres combinaciones se utilizan realmente y los hemos designado con tres identificadores.

Función	Significado
`bool setFiltroMag_Min(unsigned`	Especifica como filtro Magnification/Minification uno

`char` *`funcionFiltro`*`)`	identificado por *funcionFiltro*, que puede tener uno de los valores especificados en la tabla siguiente.

Identificadores	Significado
`POINT`	Se usa el píxel más cercano de la textura para aplicar en cada píxel del polígono. Método más rápido pero visualmente menos atractivo, las texturas aparecen pixeladas en la pantalla.
`BILINEAR`	Se usa una interpolación linear entre los píxeles cercanos. Método más usual, equilibrado en eficiencia y en resultados.
`TRILINEAR`	Se usa una interpolación lineal entre los píxeles más cercanos y el mipmap más cercano. Sólo se puede utilizar si se ha especificado mipmapping para la textura. Produce imágenes muy suaves en la pantalla pero es el método más costoso.

Funciones de Pegado de la textura

Estas funciones especifican cómo se deben mezclar la textura con el polígono al que se aplica. La función utilizada utiliza las mismas constantes de OpenGL ya que en este apartado no hay nada que añadir o simplificar. Cuando se habla de pixels del polígono, nos referimos al color de los pixels que tendría el polígono en ausencia de texturado.

Cuando se usa multitexturado, al especificar la función de pegado para una textura, se especifica la función para la unidad de texturado donde se active la textura. De esta forma se pueden conseguir muchos efectos. Por ejemplo:

- En la primera unidad de texturado `GL_REPLACE`
- En la segunda unidad de texturado `GL_MODULATE`

Con estas funciones, aparecerá la imagen como multiplicación de las dos texturas. Muy útil esta combinación para el efecto de *lightmapping*.

Función	Significado
`bool setFuncionPegado(GLint funcionPegado)`	Especifica la función que se utilizará al aplicar la textura al polígono. Los valores posibles de *funcionPegado* se especifican en la siguiente tabla.

Identificadores	Significado
GL_MODULATE	Los píxeles de la textura se multiplican con los píxeles del polígono. Si se ha definido un polígono por ejemplote color rojo, la textura aplicada con este método aparecerá con tonos rojos.
GL_DECAL	Igual que la función GL_REPLACE si la textura no posee canal alpha. Si la textura tiene color alpha, se realiza una mezcla entre los colores de la textura y los píxeles que tendría el polígono sin texturar. La mezcla de color es proporcional a la componente alpha.
GL_BLEND	Se realiza una mezcla entre los píxeles de la textura y los píxeles del polígono sin texturar.
GL_REPLACE	Reemplaza los píxeles del polígono por los píxeles de la textura.

Es muy importante conocer las diferencias entre asignar como función de pegado GL_BLEND y utilizar *blending*. En el primer caso el *blending* sólo se realiza entre los pixels de la textura con los pixels del polígono sin texturar. En el segundo caso, se realiza la mezcla entre los pixels del polígono ya texturado (con la función de pegado ya aplicada) y los pixels que ya están en el framebuffer.

"Repeating and Clamping"

Estas funciones definen el comportamiento de la textura cuando se especifican coordenadas de textura mayores que uno. La clase CTextura tiene la misma funcionalidad que OpenGL en este apartado.

Función	Significado
bool setParametroWrap(GLint funcionWrap)	Fija la función utilizada, bien GL_CLAMP o GL_REPEAT

Identificadores	Significado
GL_CLAMP	Mismo funcionamiento que OpenGL, para coordenadas de textura mayores que uno se aplica repetidamente los píxeles situados en la coordenada 1 de la textura.
GL_REPEAT	Mismo funcionamiento que OpenGL. Para una coordenada con valor e.d (e parte

	entera con e >=1 y d parte decimal) , se aplican los pixels situados en la posición 0.d

Activación de la textura

Una vez definidas todas las propiedades de la textura, ya está lista para ser utilizada. Para aplicar una textura a un polígono, sólo hay que activar la textura antes de definir la geometría del polígono (vértices y coordenadas de textura). Las coordenadas de textura también se pueden especificar con el método `coordTextura(float s, float t)` tal y como se explicó en el apartado de Multitextura. Cuando se ha terminado de aplicar la textura es conveniente desactivarla para dejar el estado de OpenGL tal y como estaba antes. Así se evita que un código posterior utilice funciones de OpenGL que haya utilizado la textura, como el *blending* por ejemplo. Cuando se utilice multitextura, se pueden activar más de una textura a la vez, teniendo en cuenta que deberían estar en una unidad de texturado diferente.

Función	Significado
`void activar(void)`	Activa la textura para ser utilizada con todos los parámetros especificados anteriormente. También activa el texturado.
`void activarID(void)`	Sólo activa el identificador de textura, por lo que no se activa ninguno de los parámetros definidos anteriormente.
`void desactivar(void)`	Desactiva la textura y cualquiera de los parámetros especificados para la textura. Desactiva también el texturado.

Valores por defecto

Si no se especifica función de pegado, mezcla... dependiendo de la imagen cargada se asignan unos valores por defecto, los que se supone que se utilizarían con ese formato de imagen. Por ejemplo, si se carga una imagen normal y se especifica que tiene un color transparente con `cargaImagenTransparente(...)`, la función de mezcla por defecto será `MEZCLA_PONDERADA` para eliminar el color transparente al dibujar el polígono. De todas formas, se recomienda fijar todos los parámetros al crear el objeto `CTextura`.

Ejemplo de uso

Cargar una imagen bmp

Para utilizar como textura una imagen BMP típica, a 24 bpp, sólo hay que realizar 4 pasos. Suponemos que la imagen está almacenada en el subdirectorio data de la carpeta del archivo ejecutable.El primer paso es crear el objeto `CTextura`. No hace falta crearlo de manera dinámica. El valor por defecto de la función de pegado es `GL_MODULATE`, y lo cambiamos a `GL_REPLACE` para poner directamente los pixels de la textura en el polígono.

```
CTextura texBitmap;
…
texBitmap = new CTextura;
texBitmap->setFuncionPegado(GL_REPLACE);
texBitmap->cargarImagen("data/pandilla.bmp");
...
texBitmap->activar();
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(...);
glTexCoord2f(1.0f, 0.0f); glVertex3f(...);
glTexCoord2f(1.0f, 1.0f); glVertex3f(...);
glTexCoord2f(0.0f, 1.0f); glVertex3f(...);
glEnd();
texBitmap->desactivar();
...
delete texBitmap;
```

Finalmente sólo hay que activar la textura antes de definir la geometría y desactivarla después. Cuando el objeto no se vaya a necesitar más, se puede liberar. No cargar texturas durante la ejecución de código crítico porque es una tarea costosa. Es preferible cargar las texturas al inicio de la aplicación por ejemplo.

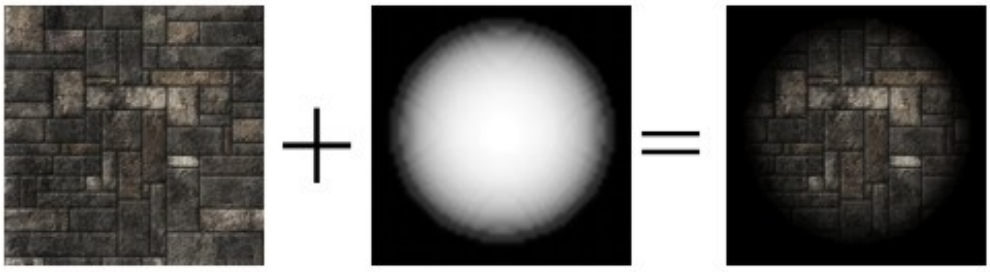
Textura con zonas transparentes

Si se escoge una imagen con canal alpha, se tiene la posibilidad de utilizar blending para realizar efectos interesantes. Uno de ellos es la eliminación de zonas transparentes. Si se fija la función de mezcla a `MEZCLA_PONDERADA`, las zonas con alpha igual a 1 reemplazarán los pixels del framebuffer y las zonas con alpha igual a 0, no serán dibujadas.

Esta imagen es un archivo TGA que tiene el canal alpha a cero en las zonas de color azul. Se puede cargar la imagen con
`cargarImagen(…)`
o también se puede hacer con
`cargarImagenTransparente()`
especificando al color azul de la imagen como color transparente. Lo único que cambia respecto al ejemplo anterior es que hay que añadir la línea
`texBitmap->setMezcla(MEZCLA_PONDERADA)`
para que se realice bien el color keying.

Ejemplo de multitextura: lightmapping

Cuando se quieren aplicar efectos de multitextura, sólo hay que cargar las imágenes en objetos `CTextura` diferentes y asignarlos a unidades de texturado diferentes. Entonces se decide que función de pegado utilizar en cada una y se activan todas las texturas involucradas en el proceso. Para generar las coordenadas de textura de cada una, utilizar el método `coordTextura(s,t)` para fijarlas. Es necesario definir las coordenadas de textura para cada una porque no tienen por que ser las mismas.

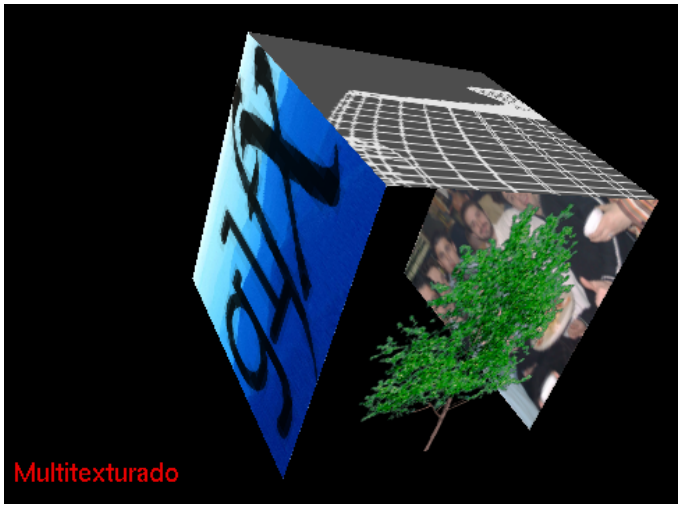


En este caso se realiza la función típica del *lightmapping*. La primera textura se aplica con GL_REPLACE y la segunda con GL_MODULATE, dando como resultado una sensación de iluminación de la textura. El código resultante sería:

```
CTextura* texMulti1, texMulti2;
…
texMulti1 = new CTextura;
texMulti1->setFuncionPegado(GL_REPLACE);
texMulti1->cargarImagen("data/terreno.gif");
texMulti2 = new CTextura;
texMulti2->setFuncionPegado(GL_MODULATE);
texMulti2->cargarImagen("data/luz.bmp");
texMulti2->setUnidadTexturado(1);
…
texMulti1->activar();
texMulti2->activar();
glBegin(GL_QUADS);
     texMulti1->coordTextura(0,0);
     texMulti2->coordTextura(0,0);
     glVertex3f(...);
     texMulti1->coordTextura(1,0);
     texMulti2->coordTextura(1,0);
     glVertex3f(...);
     texMulti1->coordTextura(1,1);
     texMulti2->coordTextura(1,1);
     glVertex3f(...);
     texMulti1->coordTextura(0,1);
     texMulti2->coordTextura(0,1);
     glVertex3f(...);
glEnd();
…
texMulti2->desactivar();
texMulti1->desactivar();
```

Demostraciones incluidas en el CD

La demostración incluida en el CD **dtextura.bpr** es un proyecto C++ Builder que muestra cómo utilizar las características del objeto `CTextura`, incluido como realizar un renderizado sobre la textura.



Efecto: Billboarding

Introducción

Generar un modelo 3D de un objeto tan complejo como lo es un árbol no es nada sencillo. Además si lo que se quiere simular realmente es un bosque entonces el tiempo de renderización se dispara. Para solucionar esto surgió hace unos 10 años la técnica del "Billboarding". *Consiste en dibujar en una escena 3D una imagen 2D* que siempre este *perpendicular a la vista del espectador*. Así la imagen tiene que rotar a medida que lo hace el espectador, porque sino se daría cuenta de que no tiene una tercera dimensión.

Explicación del Efecto

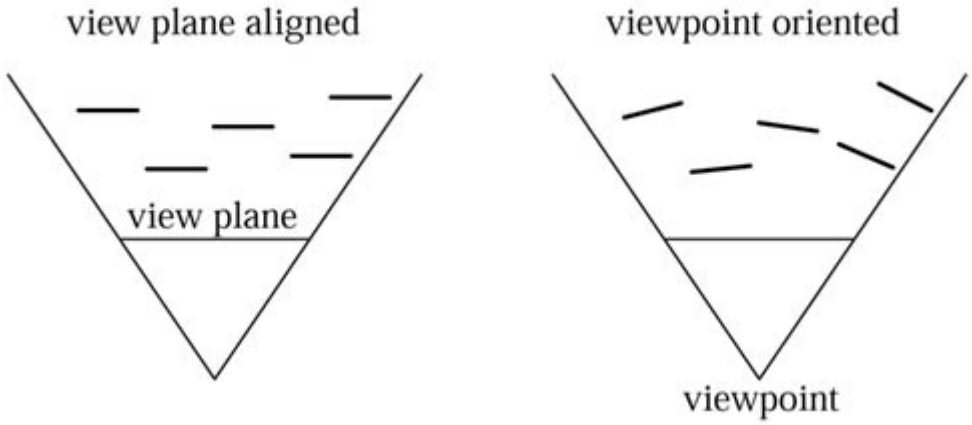
La técnica de *billboarding* es muy popular en los juegos y aplicaciones que requieren un gran número de polígonos. La lógica en la que se basa esta técnica es muy sencilla. Lo que se hace es sustituir una representación tridimensional de un objeto por su representación bidimensional, pero haciendo que el objeto siempre esté orientado, de forma que la ausencia de la tercera dimensión no se note. De esta forma, el observador no se dará cuenta de que lo que realmente está observando es una representación "plana" de un objeto tridimensional.

Utilizando esta técnica, hay que estar atento a los "contras" que se obtienen. Uno de los detalles a tener en cuenta, por ejemplo, está relacionado con las sombras. Si se utiliza una única textura para todas las representaciones del objeto, siempre se tendrá la misma sombra sobre el objeto (esté dónde esté la cámara). Este detalle puede pasar o no desapercibido, pero es una de las consecuencias del uso de esta técnica. La impresión que dará es que la luz se está moviendo a la vez que la cámara.

Aunque la principal aplicación del *billboarding* sea la de ahorrar polígonos, sustituyendo una representación tridimensional por una textura plana, no es el único uso que le podemos dar a esta técnica. Si en vez de utilizar una representación plana para el objeto a ser dibujado, usamos una representación tridimensional, esta técnica no ahorra polígonos, pero sí que se consigue que el objeto a ser dibujado esté mirando a un lugar determinado (ya sea la cámara, o un balón en un juego de fútbol, o un adversario en uno de lucha, etc.). Así pues, también se puede utilizar para "orientar" el objeto hacia algún lugar o alguna persona determinada.

Existen dos tipos de *billboarding*: el *billboarding cilíndrico* y el *esférico*. En el caso del *billboarding* esférico, no existen limitaciones sobre la orientación del objeto, mientras que en el caso del cilíndrico, se suele restringir en un eje su orientación (normalmente, el eje OY).

Además de estos dos tipos de *billboarding*, existen dos formas de afrontar el problema de la orientación del objeto:

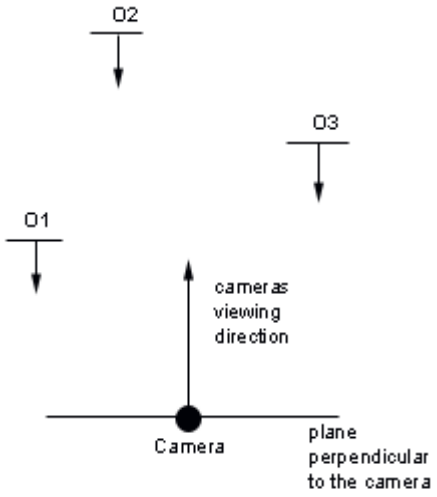


En el primer enfoque, lo que se hace es que el objeto a dibujar esté orientado con respecto del *plano de observación*, mientras que en el segundo caso, lo que se hace es orientarlo respecto del *punto de observación*. El enfoque a utilizar dependerá en cada caso.

Así pues, tendremos cuatro enfoques posibles:

(1) Billboarding esférico respecto del plano de observación.
(2) Billboarding cilíndrico respecto del plano de observación.
(3) Billboarding cilíndrico respecto del punto de observación.
(4) Billboarding esférico respecto del punto de observación.

Centrémonos inicialmente en el caso del **enfoque (1)** (Billboarding esférico respecto del plano de observación). El siguiente diagrama nos dará una imagen más clara de lo que se quiere conseguir. El punto negro representa a la cámara. El vector que sale de la cámara es la dirección actual de observación. Este vector define la orientación del plano perpendicular respecto de la dirección de observación (que es el plano con respecto al que queremos orientar nuestros objetos). Los objetos de la escena (O1..O3), han sido rotados, para que su vector de orientación frontal esté apuntando a este plano. Se asume que los objetos de la escena están posicionados en el origen local, y que están mirando en el sentido positivo del eje OZ.



Para conseguir esta transformación, hay que hacer modificaciones en la matriz de modelado. En la matriz de modelado, tenemos la siguiente estructura:

M1

a0	a4	a8	a12
a1	a5	a9	a13
a2	a6	a10	a14
a3	a7	a11	a15

El vector de la derecha, marcado en rojo, es la posición actual del origen. La submatriz 3x3 marcada en azul, contiene las operaciones de escalado y de rotación. Sustituyendo esta submatriz por la identidad, se invierten esas conversiones, haciendo que la orientación de la cámara esté alineada al eje OZ. Así pues, la nueva matriz de modelado sería:

M1

1	0	0	a12
0	1	0	a13
0	0	1	a14
a3	a7	a11	a15

Con esta transformación, como se puede ver, se deshacen todas las operaciones de escalado sobre el objeto. Así pues, si se quiere dibujar el objeto escalado, lo que habrá que hacer es que, después de haber realizado esta transformación, y antes de dibujar el objeto, habrá que "reescalarlo", para obtener las dimensiones deseadas. Otro de los "contras" de este método es que cuando el observador mire hacia arriba o hacia abajo, el objeto dibujado se inclinará hacia delante o hacia atrás (respectivamente). Esto se debe a que utilizamos el plano de observación de la cámara, y este plano se ve modificado al mirar hacia arriba y hacia abajo. Este tipo "efecto secundario" se elimina en los casos en los que no se utiliza un enfoque respecto del *plano de observación*, si no

respecto del *punto de observación*. El plano de observación sí que se ve modificado al mirar hacia arriba y hacia abajo, mientras que el punto de observación se mantiene constante, sea cual sea la dirección de observación.

Ahora pasaremos a explicar el **enfoque (2)** (Billboarding cilíndrico respecto del plano de observación). Para esto, tendremos que explicar con un poco más de detalle qué significa la submatriz superior 3x3 de la matriz de modelado. El primer vector (todos los vectores están formados por columnas), es el vector "*Right*". Este es el vector que define el eje OX de la cámara. El segundo vector recibe el nombre de "*Up*" y es el vector del eje OY de la cámara. El tercer vector, como es de esperar, es el del eje OZ de la cámara, y recibe el nombre de "*LookAt*". En el caso anterior, lo que se conseguía sustituyendo esta submatriz por la identidad, era que los 3 ejes de la cámara tuviesen la misma orientación que los 3 ejes del sistema de coordenadas de OpenGL.

Como ya hemos dicho antes, en el caso del billboarding cilíndrico, lo que se hace es que se restringe la rotación en uno de esos ejes (normalmente el eje OY de la cámara). Con esto eliminamos uno de los "efectos secundarios" que teníamos en el caso esférico planar. Al restringir la rotación en el eje OY, cuando el observador mire hacia arriba o hacia abajo, el objeto no sufre transformación alguna y, por lo tanto, ya no se inclinará hacia delante o hacia atrás, como ocurría en el caso anterior. Así pues, la matriz de modelado que queremos obtener para implementar esta aproximación será la siguiente:

M1

1	a4	0	a12
0	a5	0	a13
0	a6	1	a14
a3	a7	a11	a15

Como se puede observar, se sustituyen los vectores "*Right*" y "*LookAt*", pero se mantiene intacto el vector "*Up*".

Uno de los "contras" que tiene este enfoque está relacionado con las aplicaciones que pueden sobrevolar los objetos. En este caso, lo que ocurrirá es que el objeto dibujado se irá haciendo cada vez más pequeño, a medida que nos aproximemos a él. Cuando sobrevolemos el objeto, la ilusión del objeto tridimensional se romperá y, a medida que nos alejemos del objeto, éste irá recuperando su forma y tamaños iniciales.

Hay enfoque intermedio que se puede aplicar para estas dos primeras transformaciones que hemos visto. En vez de realizar las transformaciones sobre la matriz de modelado, lo que se puede hacer es realizar los cálculos necesarios para obtener las coordenadas de los extremos del objeto a dibujar. El resultado final es el mismo que en los casos anteriores (existen tanto el enfoque cilíndrico como el esférico para esta solución), pero en este caso se obtiene de forma más rápida, ya que la matriz

de modelado sólo se obtiene una vez por cada "frame" dibujado, mientras que en el caso anterior, se obtenía una vez por cada objeto dibujado. En resumen, este método es más rápido que el anterior, pero más difícil de implementar.

Veamos que necesitamos para realizar estas transformaciones. Los vértices del billboard se obtienen utilizando los vectores "*Up*" y "*Right*" que invierten las orientaciones de la matriz de modelado. Estos vectores se pueden obtener de la inversa de la submatriz 3x3 de la matriz de modelado. Por suerte, la inversa de la matriz de modelado (y, por lo tanto, de su submatriz superior 3x3), es la traspuesta. Así pues, la inversa de la submatriz superior 3x3 será:

$$M^{-1} = \begin{bmatrix} a0 & a1 & a2 \\ a4 & a5 & a6 \\ a8 & a9 & a10 \end{bmatrix}$$

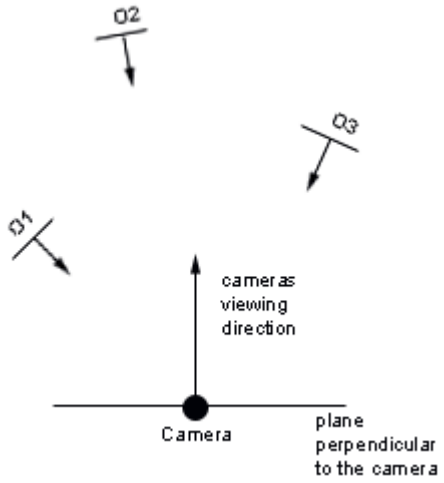
Teniendo esta matriz, obtener los vectores "*Right*" y "*Up*" es directo, ya que son la primera y segunda columna respectivamente. En el caso del *billboarding* cilíndrico, el vector up no se necesita, ya que será *up = {0, 1, 0}*. Ejemplos de la aplicación de este método se verán en la sección ***Formas de Uso y Ejemplos***.

Pasemos al **enfoque (3)** (Billboarding cilíndrico respecto del punto de observación). El efecto que queremos obtener en este caso, es que el objeto a dibujar no esté mirando en el mismo plano de observación que la cámara, si no que esté mirando directamente al punto en el que esté la cámara.

Este método es muy utilizado. Aquí tenemos un ejemplo de uso, para un simulador de aviones:

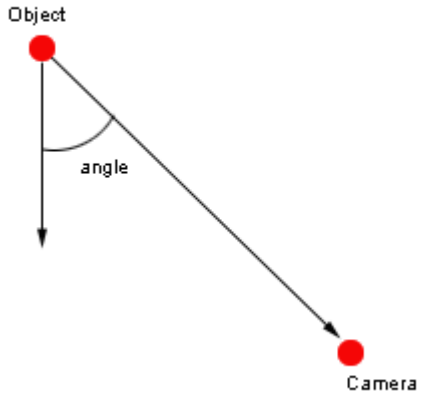
Podemos ver que las nubes están todas apuntando a la posición en la que se encuentra el avión.

En el caso del *billboarding* cilíndrico no se hace que mire directamente a la cámara, pero sí al punto en el que estaría la cámara si se encontrase en el mismo plano XZ que el objeto (quiere esto decir que ignoramos la posición de la cámara con respecto al eje OY, y nos centramos en las transformaciones sobre los otros dos ejes). El efecto que se quiere obtener es el siguiente:



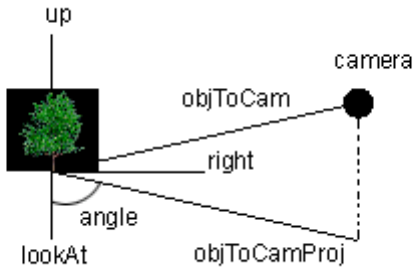
Se puede ver ahora que los objetos O1..O3 apuntan directamente a la posición en la que se encuentra la cámara, y no al plano que define la dirección de observación de ésta. Para poder hacer esta transformación, necesitamos las coordenadas de la cámara y las del objeto a dibujar en coordenadas de OpenGL

Para orientar el objeto con respecto al punto en el que se encuentra la cámara, lo que tenemos que hacer es rotar el objeto sobre el vector "*Up*", para hacer que su vector "*LookAt*" sea el vector desde el objeto a la cámara. Lo que queremos obtener esta representado en la siguiente figura:



El primer vector (el que apunta hacia abajo) es la supuesta orientación del objeto, el vector "*objACam*" es la orientación que queremos obtener y "*angle*" es el ángulo definido por estos dos vectores.

La condición inicial que tendremos es la siguiente:



Lo que tenemos que hacer para poder realizar la rotación es lo siguiente:

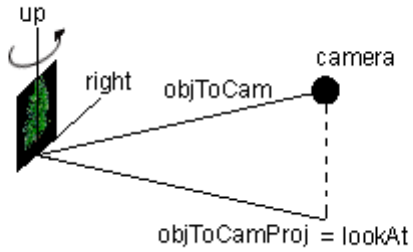
(1) Proyectamos el vector "*objACam*" sobre el plano XZ (esto quiere decir que la componente Y del vector "*objACamProy*", proyección del vector "*objACam*" sobre el plano XZ, será cero).

(2) Normalizamos el vector "*objACamProy*".

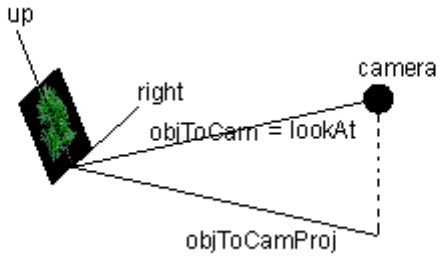
(3) Calculamos el coseno, haciendo el producto escalar del vector "*LookAt*" y el vector "*objACamProy*" (al estar normalizados, el cálculo es directo).
Hacemos el producto vectorial de "*LookAt*"x"*objACamProy*", para así obtener un vector "*upAux*" (perpendicular a los dos vectores anteriores), sobre el que efectuar la rotación.

(4) Rotamos el ángulo calculado (por medio de la inversa del coseno) sobre el vector "*upAux*" nuevo.

La nueva situación que tendremos, será la siguiente:

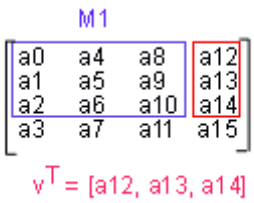


El caso del **enfoque (4)** (Billboarding esférico respecto al punto de observación) es muy similar al anterior. Lo que se hace en el caso del *billboarding* esférico es, además de todas las transformaciones que se hacen en el caso cilíndrico, realizar una rotación más, para hacer que el objeto apunte directamente a la cámara, y no sólo a su proyección sobre el plano XZ. La situación final será la siguiente:



NOTA SOBRE EL CÁLCULO DE LAS COORDENADAS DEL MUNDO

Como hemos dicho, para poder realizar todos estos cálculos, lo que necesitamos son las posiciones del objeto y de la cámara en coordenadas del mundo. Para poder obtenerlas, hay un método muy sencillo. Este método está basado, de nuevo, en la matriz de modelado.



El vector "v" es la posición del origen en coordenadas de la cámara. Para poder obtener la posición en coordenadas reales, lo que tenemos que hacer es lo siguiente:

(1) Obtenemos el vector v^T.

(2) Obtenemos la inversa de la matriz M1 (que, como sabemos, es igual a su traspuesta).

(3) Multiplicamos $M1^T$ x v^T, lo que nos da una posición.

(4) A esta posición le tendremos que sumar la posición de la cámara (en coordenadas "del mundo").

Para poder hacer estos cálculos, tenemos la siguiente función:

```
localToWorld (float* cam, float* worldPos)
```

El modo de uso de esta función se explica en la sección **Funciones**. A grandes rasgos, lo que hace esta función es convertir el origen de coordenadas actual en coordenadas de OpenGL, pasándole la posición actual de la cámara. Necesitamos saber la posición actual de la cámara, ya que las coordenadas actuales están definidas con respecto del sistema de coordenadas de la cámara.

Archivos

Este efecto gráfico está implementado en un objeto de tipo `Billboarder`. La definición de esta clase se encuentra en la unidad "billboarding.h".

Funciones

`Billboarder()`: Constructora del objeto `Billboarder` sin parámetros. Esta constructora utiliza por defecto el billboarding planar cilíndrico.

`Billboarder(GLenum modo)`: Constructora del objeto `Billboarder`, en la que se define el tipo de billboarding a seguir. El valor de *modo* puede ser:

- `BILLBOARD_PLANAR_CILINDRICO`
- `BILLBOARD_PLANAR_ESFERICO`
- `BILLBOARD_PUNTUAL_CILINDRICO`
- `BILLBOARD_PUNTUAL_ESFERICO`

`bool activar()`: Esta función activa el billboarding. El modo en el que se activa el billboarding habrá sido dado con anterioridad (en la construcción del

Billboarder se define un tipo de billboard, que puede ser cambiado mediante llamadas a la función `setModo(GLenum modo)`). Esta función sólo funcionará correctamente (y devolverá **true**) si el modo de billboard seleccionado es de tipo PLANAR. En caso contrario, fallará (y devolverá **false**).

`bool activar(float* cam, float* worldPos)`: Esta función activa el billboarding. El modo en el que se activa el billboarding habrá sido dado con anterioridad (en la construcción del *Billboarder* se define un tipo de billboard, que puede ser cambiado mediante llamadas a la función `setModo(GLenum modo)`). Esta función funciona correctamente en todos los casos. La única diferencia, es que en los casos PLANARES, se ignoran los parámetros dados, ya que no son necesarios. Los parámetros dados son la posición de la cámara y la posición del objeto a dibujar (en coordenadas del sistema de OpenGL).

`bool activar(GLenum modo)`: Esta función activa el billboarding en el modo seleccionado por el parámetro *modo*. Esta función sólo funcionará correctamente (y devolverá **true**) si el modo de billboard seleccionado es de tipo PLANAR. En caso contrario, fallará (y devolverá **false**). El valor de este parámetro puede ser:

- `BILLBOARD_PLANAR_CILINDRICO`
- `BILLBOARD_PLANAR_ESFERICO`
- `BILLBOARD_PUNTUAL_CILINDRICO`
- `BILLBOARD_PUNTUAL_ESFERICO`

`bool activar(GLenum modo, float* cam, float* worldPos)`: Esta función activa el billboarding en el modo seleccionado por el parámetro *modo*. Esta función funciona correctamente en todos los casos. La única diferencia, es que en los casos PLANARES, se ignoran los parámetros dados, ya que no son necesarios). El parámetro *modo* puede tomar los mismos valores que en el caso anterior. El resto de los parámetros dados son la posición de la cámara y la posición del objeto a dibujar (en coordenadas del sistema de OpenGL).

`GLenum getModo()`: Esta función devuelve el modo de billboarding usado. Los valores posibles son los siguientes:

- `BILLBOARD_PLANAR_CILINDRICO`
- `BILLBOARD_PLANAR_ESFERICO`
- `BILLBOARD_PUNTUAL_CILINDRICO`
- `BILLBOARD_PUNTUAL_ESFERICO`

`void setModo (GLenum modo)`: Esta función selecciona el modo de billboarding a usar. El valor del parámetro *modo* puede tomar los mismos valores que en la función anterior.

`void localToWorld(float *cam, float *worldPos)`: Con esta función se transforman las coordenadas de la cámara a coordenadas "del mundo". Este método toma como punto a convertir el origen de coordenadas que tenemos en la matriz de modelado. Se le pasa como parámetro un vector con las coordenadas de la cámara (**cam*), y devuelve otro vector con las coordenadas "del mundo" (**worldPos*) del origen de coordenadas que se encuentra en la matriz de modelado.

`void getRightVector(float *right)`: Esta función devuelve el vector "*right*" (escrito en el vector pasado como parámetro) necesario para hacer las transformaciones de los vértices el objeto a dibujar, en el caso del billboarding cilíndrico (ya que el vector up es fijo, y vale [0, 1, 0]).

`void getUpRightVector(float *up, float *right)`: Esta función devuelve los vectores "*up*" y "*right*" (escritos en los vectores pasados como parámetros) necesarios para hacer las transformaciones de los vértices el objeto a dibujar.

`void desactivar()`: Función que deshace las transformaciones realizadas por cualquiera de los métodos de billboarding anteriores.

Formas de Uso y Ejemplos

La forma de uso de este efecto es muy sencilla para todos los casos. El primer paso en todos los casos es el mismo (crear el objeto *Billboarder*). El código es bastante trivial:

```
Billboarder* bb = new Billboarder();
// Billboarder* bb = new Billboarder(tipo);
```

En el caso de la segunda llamada (la que está entre comentarios), el valor de tipo tendría que ser alguno de los dados en la sección **Funciones**.

Una vez creado el objeto, y seleccionado el tipo (recordemos que en el caso de la constructora sin parámetros se establece por defecto el billboarding planar cilíndrico), tenemos varias posibilidades. En el caso de los billboarding planares, un código de ejemplo sería el siguiente:

```
// Primero activamos el billboarding
bb->activar();
// Escalar el objeto (si es  necesario)
glScale(x,y,z);
drawObject();  // dibujamos el objeto
bb->desactivar();   // y desactivamos el billboarding
```

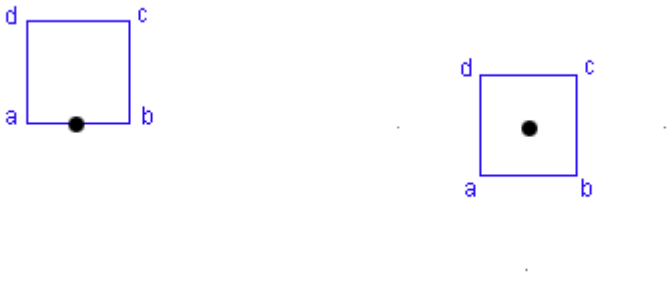
En el caso de los otros tipos de billboarding (billboarding cilíndrico y esférico con respecto del punto de visión), las llamadas también son similares.

```
// tenemos que tener la posición de la cámara
cam = {x, y, z};
float* pos = "posición actual";
// convertimos las coordenadas de cámara a
// coordenadas de OpenGL
bb->localToWorld (cam, pos);
// activamos el billboarding según el modo elegido
bb->activar (cam, pos);
drawObject();       // dibujamos el objeto
bb->desactivar();   // desactivar el billboarding
```

Si el modo de billboard que tenemos elegido es de tipo planar, la llamada a activar también puede ser:

```
// no necesitamos la posición de la camara ni la
// posición actual.
bb->activar(0,0);
```

El caso del cálculo manual de los extremos del objeto a dibujar, tiene varias aproximaciones. Para calcular estos extremos, necesitaremos los vectores "*up*" y "*right*" (en el caso del billboard cilíndrico sólo se necesita el vector "*up*", ya que el vector "*right*" es igual a *right = {0, 1, 0}*). Una vez que los tengamos, el cálculo de los extremos dependerá del enfoque que le demos. Aquí demostraremos dos posibilidades, habiendo muchas más:



La primera figura asume que el origen está en el lado inferior de la imagen, y que ésta se distribuye de la misma forma a izquierda y derecha del origen (este enfoque se suele utilizar para dibujar objetos al nivel del suelo). Los cálculos necesarios para el primer ejemplo serán los siguientes:

```
float* up, *right;
bb->getUpRightVector (up, right);
// en el caso del billboard cilíndrico,
// llamada será bb->getRightVector (right)
// y el vector up = {0, 1, 0};
// las siguientes líneas están en “pseudocódigo”,
// ya que las coordenadas de cada punto
//  constan  de  3  dimensiones,  y  se  tendrían  que
calcular
// una a una (es decir, tres líneas de
// código por punto, una por cada componente).
a = centro - right * (tam / 2);
b = centro + right * (tam / 2);
c = centro + right * (tam / 2) + up * tam;
d = centro - right * (tam / 2) + up * tam;
```

En el segundo caso, lo que tenemos es el origen en el centro de la imagen. Los cálculos para este segundo ejemplo, serán los siguientes:

```
float* up, *right;
bb->getUpRightVector (up, right);
// en el caso del billboard cilíndrico, la
// llamada será bb->getRightVector (right)
// y el vector up = {0, 1, 0}

// las siguientes líneas están en “pseudocódigo”,
// ya que las coordenadas de cada punto
// de calcularán una a una (es decir, tres
// líneas de código, una  por cada componente del
// punto).
a = centro - (right + up) * tam;
b = centro + (right - up) * tam;
c = centro + (right + up) * tam;
d = centro - (right - up) * tam;
```

Animación en 2D. Sprites.

Introducción

Debido a la mejora que ha experimentado el hardware de aceleración 3D, y la creciente demanda de realismo en videojuegos y aplicaciones gráficas en general, las técnicas de programación en 2 dimensiones son cada vez menos usadas como enfoque gráfico general. Aun así, esta técnica presenta ventajas y aplicaciones que pueden ser usadas en multitud de ocasiones, debido a factores como su rendimiento. Una aplicación gráfica puede usar **sprites** (objetos bidimensionales animados que representan entidades) para:

- Simular efectos de '*scrolling*', incluso de varios niveles (técnica conocida como '*parallax*').
- Representar objetos animados alejados o pequeños, de tal modo que al espectador no le sea fácil darse cuenta de la bidimensionalidad de la entidad.
- Dibujar carteles en movimiento (estilo vallas publicitarias), imágenes en televisiones, etc.

Las posibles usos son muy variados y casi cualquier aplicación puede beneficiarse de usar animación en 2D.

Consideraciones iniciales.

Ya hemos visto que a pesar de parecer una técnica obsoleta, queda probada su utilidad, sino como estilo de visualización de la aplicación, sí como complemento de una visualización más compleja en 3D. Debemos discutir ahora el enfoque de implementación para que este tipo de efectos quede integrado en una arquitectura, tanto hardware como software, orientada totalmente hacia la manipulación de gráficos tridimensionales, y que adolece de ciertas carencias (como un acceso sencillo a píxeles individuales de una imagen o 'superficie', como se las llama en la bibliografía, o un '*bit blitting*' eficiente), que no existían cuando la programación gráfica era exclusivamente bidimensional.

Debemos pues hallar la forma de aprovechar la aceleración hardware del procesamiento bidimensional para crear un sistema de sprites y bitmaps eficiente. Esto, que puede parecer contradictorio en un principio, se convierte en obvio si pensamos que todos los APIs tridimensionales proporcionan un soporte muy eficiente para el uso de texturas. Por lo tanto implementaremos un sistema de gráficos bidimensionales basado en polígonos planos texturados. Así conseguiremos una perfecta integración de gráficos 3D y 2D, incluso podremos construir

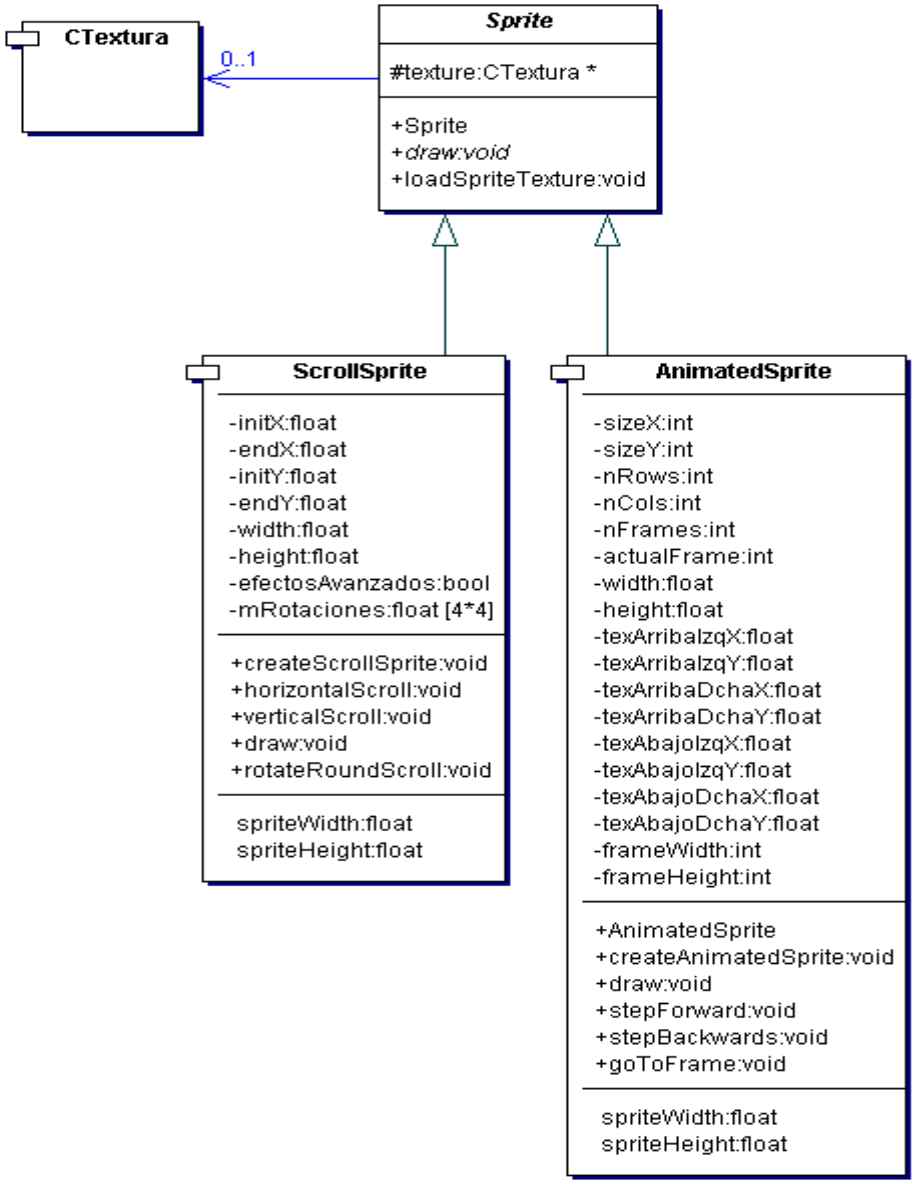
aplicaciones gráficas exclusivamente bidimensionales que se verán aceleradas por el hardware de procesamiento tridimensional.

Arquitectura del sistema

Dividiremos el sistema en dos partes bien diferenciadas por su funcionalidad y posibles usos.

- `AnimatedSprites`: Esta clase representará los típicos sprites o entidades que pueden tener una animación basada en 'frames' o diferentes imágenes que al sucederse crean la sensación de animación.

- `ScrollSprite`: Esta clase representará imágenes que pueden ser movidas horizontal o verticalmente, o rotadas, para poder crear efectos de desplazamiento o 'scrolling' (útiles para cielos, suelos, carteles móviles, cintas transportadoras, etc.).



Ambas clases heredarán de una clase padre `Sprite` que proporcionará de forma abstracta dos únicos métodos:

- `Sprite::draw()`: Método abstracto que debe ser redefinido y que pintará el Sprite en el contexto de OpenGL.
- `Sprite::loadSpriteTexture(CTextura* tex)`: Asigna al sprite una textura.

Implementación.

Ambos tipos de Sprite, para conseguir los objetivos antes mencionados de ser soportados por el hardware actual y ser eficientes, serán implementados usando polígonos planos texturados, en concreto del tipo 'quad' (rectángulos). De este modo no serán una sobrecarga para el sistema debido al bajo número de triángulos que los compondrán (dos en concreto), y podremos utilizar con ellos otras técnicas desarrolladas así como la potencia del objeto `CTextura`. Por lo tanto lo único que resta por hacer es el texturado del polígono.

Al usar el objeto `CTextura`, podemos obviar la parte de inicialización de la imagen y otras cuestiones como la función de pegado, transparencias, *blending*, etc. Únicamente debemos preocuparnos por como mapear la textura sobre el rectángulo. Para ello debemos conocer el sistema de coordenadas de textura de OpenGL, y manipularlo de forma adecuada, tal y como puede verse en las funciones de dibujado de los sprites.

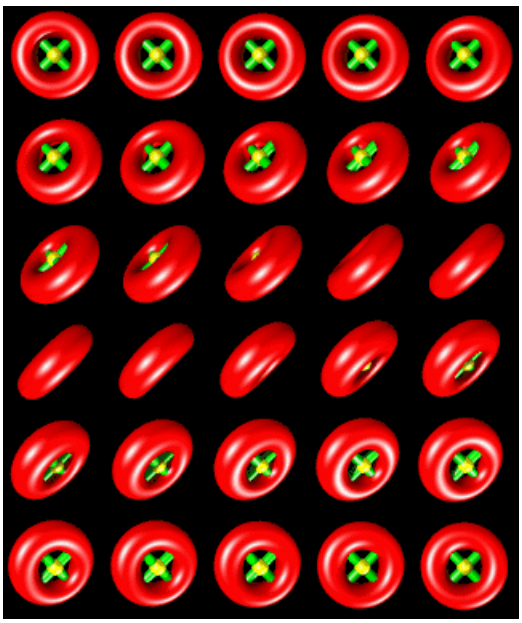
Por último debemos observar que se puede ampliar la cantidad de efectos aplicables a una textura mediante el uso del modo avanzado de mapeado que utiliza la matriz de OpenGL `GL_TEXTURE`. Manipulándola al igual que se manipulan el resto de matrices (pero considerando el sistema de coordenadas especial de las texturas) podemos crear efectos avanzados tales como rotaciones, escalados y otras deformaciones.

Guía de uso

Lo más importante a la hora de utilizar sprites es elegir o crear correctamente la textura que los representará. En el caso de un `ScrollSprite` este proceso es sencillo, sólo se debe tener en cuenta que es posible que para crear el efecto deseado se necesite una textura de tipo '*tileable*', es decir, cuyos bordes encajen por la parte en la que se hará la transición debida al movimiento.

En el caso de un `AnimatedSprite`, la textura debe ser ligeramente más compleja: Se debe crear una que contenga todos los frames de animación posibles para el sprite, preferiblemente ordenados según como serán mostrados (para

aprovechar las funciones que proporciona la clase) y situados en forma de matriz uni o bidimensional. Por ejemplo:



En la figura anterior observamos los frames de animación de un sprite con forma de donut giratorio. El sprite tendrá 6 filas * 5 columnas de frames, haciendo un total de 30 frames. Estos datos debemos pasárselos al objeto para que sea capaz de dibujarse correctamente.

La interfaz de uso que presentan ambas clases es la siguiente:

AnimatedSprite

- `void createAnimatedSprite(int sizeX, int sizeY, int nRows, int nCols, int initFrame)`. Esta función se encarga de inicializar los parámetros internos de un objeto sprite previamente creado. Debe ser llamada antes de intentar dibujarlo. *`sizeX, sizeY`* son el tamaño en píxeles de la textura que representa el sprite y que será cargada mediante `loadSpriteTexture(...)`. *`nRows, nCols`* son el número de filas y columnas de la matriz de frames. *`initFrame`* es el frame inicial (comenzando a numerarlos desde cero).

- `void setSpriteWidth(float width)`
- `void setSpriteHeight(float height)`. Indican la anchura y altura del sprite en el sistema de OpenGL (es decir, las dimensiones del QUAD que contendrá la textura).

- `void draw()`: Dibuja el sprite en el contexto de OpenGL, siendo el punto (0,0) el centro del rectángulo que representa el sprite.

- `void stepForward()`: Avanza en un frame la animación.

- `void stepBackwards()`: Retrocede en un frame la animación.

- `void goToFrame(int frame)`: Va a un frame en concreto. Los frames se numeran comenzando por el cero y avanzando de izquierda a derecha(por columnas) y después de arriba abajo (por filas).

`ScrollSprite`

- `void createScrollSprite(float numX, float numY)`: Esta función se encarga de inicializar los parámetros internos de un objeto sprite previamente creado. Debe ser llamada antes de intentar dibujarlo. *`numX`* es el número de veces que se repetirá la textura a lo ancho del rectángulo en un momento dado. Es decir, si *`numX`* es 1 la textura encajará exactamente en el rectángulo, si es 2 se aparecerá dos veces seguidas, si es 0.5 sólo se verá la mitad, etc. *`numY`* es lo mismo que el anterior para la vertical.

- `void setSpriteWidth(float width)`
- `void setSpriteHeight(float height)`. Indican la anchura y altura del sprite en el sistema de OpenGL (es decir, las dimensiones del QUAD que contendrá la textura).

- `void horizontalScroll(float offset)`: Avanza la textura horizontalmente la cantidad indicada por *`offset`*.

- `void verticalScroll(float offset)`: Avanza la textura verticalmente la cantidad indicada por *`offset`*.

- `void draw()`: Dibuja el sprite en el contexto de OpenGL, siendo el punto (0,0) el centro del rectángulo que representa el sprite.

- `void rotateRoundScroll(float angle, float x, float y)`: Gira la textura la cantidad indicada por *`angle`* (en grados) alrededor del punto indicado por *`x`* e *`y`*, según el sistema de coordenadas interno del sprite (en el que el punto (0,0) es el centro de éste).

Ejemplo

En el ejemplo incluido en el CD observamos diversos modos de uso de las clases de sprites.

También podemos observar dos consecuencias de la implementación elegida:

- Si necesitamos diversos sprites con la misma textura, no es necesario duplicarla ya que todos pueden compartir el mismo objeto `CTextura` sin que interfieran.

- Podemos también usar una sola textura para representar los frames de dos sprites diferentes. Para ello debemos calcular los parámetros de forma que para cada uno de ellos los frames que pertenezcan al otro queden incluidos en los suyos, pero luego debemos tener la precaución de no dibujarlos.

Ambas nos proporcionan formas de ahorrar memoria de video. Por último debemos destacar que los sprites suelen usarse en conjunción con las técnicas de *billboarding* y *mipmapping*. La primera proporciona sensación de tridimensionalidad al no dejar que el observador se percate de la naturaleza 2D del objeto, mientras que la segunda evita que los sprites aparezcan pixelados cuando se acercan demasiado al observador, a la vez que evitan efectos de interferencia en las imágenes al alejarse.

Carga de objetos de modeladores externos. Mesh loading.

Introducción

Frecuentemente en el desarrollo de juegos o aplicaciones con contenidos gráficos se presenta la necesidad de utilizar objetos complejos que son difícilmente creados mediante las primitivas de OpenGL o escribiendo listas de vértices a mano. En estas ocasiones interesa el poder cargar objetos generados mediante programas modeladores, mediante los cuales se pueden generar modelos complejos de una forma amigable e intuitiva.

Para la representación de estos modelos cada programa posee sus propios formatos, que muchas veces son ampliamente utilizados, como en el caso del 3DStudio, que utiliza un formato de codificación binario basado en '*chunks*' o contenedores que organizan jerárquicamente las propiedades de los objetos que conforman el modelo. Este formato es potente pero no es sencillo para su utilización y su estructura es compleja.

Otras compañías utilizan formatos menos potentes, pero más sencillos y con una especificación portable y abierta. Es el caso del estándar de representación de modelos mediante ficheros '.obj'. Este formato consiste en archivos de texto que definen listas de vértices, caras, normales y coordenadas de textura. La sencillez de su especificación y el hecho de ser sencilla de leer y editar directamente, la hacen un formato ideal para el intercambio de modelos entre aplicaciones (de hecho la mayoría soporta este formato) y para desarrollar las principales ideas de carga de mallas. A partir de aquí podemos utilizar la misma arquitectura y clases básicas para permitir cargar de nuevos tipos de archivo.

Arquitectura básica

Se definen dos partes en el desarrollo de la funcionalidad necesaria:

`MeshLoader`: Un objeto que se encargue de leer el archivo y crear a partir de sus especificaciones el modelo que luego será mostrado. Este objeto debe conocer tanto la estructura y semántica del archivo cuyo contenido quiere leer, como la forma de crear un modelo a partir de estos datos.

`C3DModel`: Un objeto que representara el modelo en cuestión, creado a partir de los datos leídos por el `MeshLoader` y pueda ser dibujado en un contexto de OpenGL.

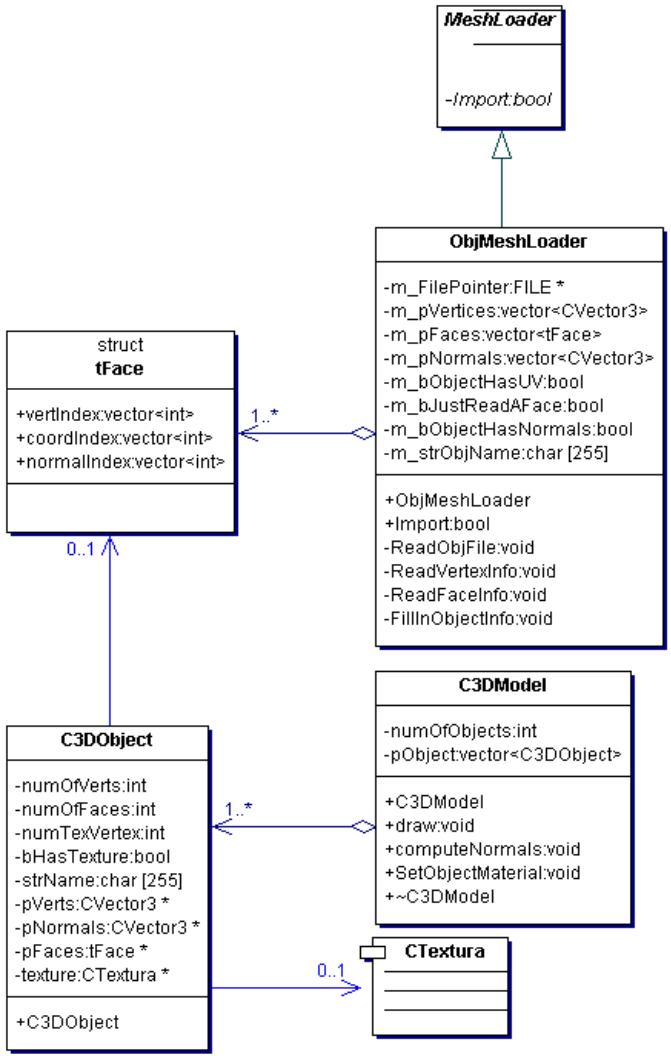
Arquitectura de MeshLoader

Se opta por desarrollar una jerarquía que prevea una posterior ampliación de los tipos de archivos legibles. De este modo se define una clase básica `MeshLoader`, abstracta, que define un único método virtual para leer el archivo y crear el modelo, que las clases derivadas deben implementar para leer un tipo de archivo en concreto.

A partir de ahí definimos una clase hija de ésta que implemente la capacidad de leer archivos '.obj'. Básicamente esta clase implementa un *parser* del archivo que luego es capaz de rellenar el modelo correctamente. De este modo la jerarquía queda como indica la figura. A partir de aquí podemos definir nuevas clases que sean capaces de leer otros tipos de archivo, como '.3DS'. '.MD2', etc.

Arquitectura de C3Dmodel

Para la implementación del objeto `C3DModel` se utilizaron tipos de datos sencillos que no ocasionaran una sobrecarga excesiva en las operaciones típicas, y suficientemente expuestas como para que la aplicación pueda acceder a los datos básicos que la componen para poder realizar operaciones avanzadas. Por estos dos motivos se elige representar el modelo como estructuras de datos cuyos miembros pueden ser accedidos y modificados. Esto puede parecer contrario al modelo de programación orientada a objetos, pero posibilita que el objeto pueda ser utilizado más que como un simple contenedor del modelo, al resultar sencillo acceder a los objetos que lo componen, listas de vértices, etc. El diagrama de clases queda como indica la figura.



Especificación del formato '.OBJ'

Este formato, como ya se ha dicho, se representa en ficheros de texto y una sintaxis basada en etiquetas. Las etiquetas comienzan una línea, y les siguen sus atributos. Las posibles etiquetas son muchas, pero sólo utilizaremos un subconjunto de ellas, que consideramos relevante para la generación de modelos. Son las siguientes:

- **v x y z***:* Define un vértice. El orden en el que son definidos los vértices determina su índice. x, y y z son las coordenadas espaciales del vértice.
- **vn x y z**: Define una normal. Mismas consideraciones que para el vértice.
- **vt u v**: Define una coordenada de textura. u y v son las coordenadas de texturado.

- **f** : Indica que se va a definir una cara. Dependiendo del objeto las caras pueden tener tres formatos:
  - **f v1 v2 v3 [v4 ...]** si el objeto no tiene coordenadas de textura ni normales.
  - **f v1/c1 v2/c2 v3/c3 [v4/c4 …]** si el objeto tiene coordenadas de textura pero no normales.
  - **f v1//n1 v2//n2 v3//n3 [v4//n4 …]** si el objeto tiene normales pero no coordenadas de textura.
  - **f v1/c1/n1 v2/c2/n2 v3/c3/n3 [v4/c4/n4 …]** si el objeto tiene definidas coordenadas de textura y normales.

En todos los casos v1,v2 ... representan los índices de los vértices que forman la cara; c1, c2... los índices de las coordenadas de textura de ese vértice y n1, n2 ... la normal de ese vértice.

Estos índices comienzan en 1 y van seguidos y sin repetición para todos los objetos que forman el modelo. Como vemos las caras pueden estar formadas por cualquier numero de vértices (todas el mismo número). Esto hace que los polígonos que forman el modelo sean triángulos, cuadriláteros o polígonos en general. Se debe hacer distinción ya que OpenGL optimiza el dibujado de triángulos y quads. El utilizar caras de un tipo u otro depende del programa que genere el archivo .obj, pero lo usual es utilizar estos dos tipos de polígonos más frecuentemente.

- **g Nombre**: Indica el nombre del objeto que se está definiendo. Los archivos .obj guardan el modelo como objetos separados, que se pueden cargar de forma individual y asignarles propiedades específicas tales como el material.

Implementación

La implementación de este sistema se reduce a la traducción del archivo a vértices y caras, y su posterior almacenamiento en el modelo. Por ello los interfaces que vamos a exigir de estas clases para su uso externo van a ser muy sencillas. Por un lado, necesitaremos que los *loaders* provean la funcionalidad de leer el fichero y cargar el modelo:

```
void virtual MeshLoader::Import(char *strFileName,
C3Dmodel *model);
```

Esta clase será abstracta (un interfaz), para que sea redefinida por las clases que implementen la traducción de los formatos de archivo concretos. Por otro el modelo debe ser capaz de dibujarse:

```
void C3DModel::draw();
```

Proporcionaremos métodos para calcular las normales (debido a que hay formatos de archivo que no las definen explícitamente) y para asignar materiales a objetos componentes (ya que los ficheros .obj permiten proporcionar coordenadas de mapeado pero no la textura o material del objeto):

```
void C3DModel::computeNormals();
void C3DModel::SetObjectMaterial(CTexture* tex, int
materialID);
```

Optamos por hacer que los *loaders* sean clases amigas de `C3DModel`. De este modo no necesitamos definir interfaces complejos entre ellas que pueden entorpecer su uso, haciéndolo más complejo.

Para la implementación del modelo queremos reflejar la organización jerárquica que existe entre modelo y los objetos que lo componen, por lo tanto definimos clases separadas para ambos. En la clase `C3DModel` representaremos la información general del modelo (principalmente los objetos que lo componen), mientras que en los `C3DObject` guardaremos las características propias de cada objeto: geometría, materiales, etc.

Para ello utilizaremos la clase de la STL (Standard Template Library) para almacenar las listas de vértices, normales y coordenadas de textura. También utilizaremos vectores para almacenar la lista de índices de vértices, normales y coordenadas de textura de los componentes de una cara. De este modo podemos hacer que las caras tengan cualquier número de vértices. Por último delegaremos la labor del texturado en la clase `CTextura`.

El método para calcular las normales de los vértices utiliza un sencillo algoritmo que primero computa la normal de cada cara (usando productos vectoriales) y a continuación calcula las normales de los vértices haciendo la media de las normales de las caras en las que participa ese vértice.

Para pintar el modelo recorremos los objetos que lo forman. Dentro de estos recorremos las listas de caras, desreferenciando los índices a vértices, coordenadas de textura y normales, y se los pasamos a OpenGL con los comandos adecuados.

Por último debemos implementar la clase que va a concretar el `MeshLoader`, y que será la encargada de leer los modelos en formato .obj. Esta clase será `ObjMeshLoader`, que hereda públicamente de `MeshLoader` y se encarga de redefinir el método 'Import'.

La implementación resulta sencilla, tan sólo hay que tener en cuanta los formatos de cada línea (que pueden variar dependiendo de las características del objeto en cuestión, en especial las caras.)

Guía de uso

Funciones

Para utilizar la clase `C3DModel` que representa el modelo cargado se utilizan las siguientes funciones:

`C3DModel::draw()`: dibuja el modelo. Este dibujado se hace sobre las proporciones del modelo cargado, y suele ser necesario un escalado para adaptarlo a las dimensiones de la escena.

`C3DModel::computeNormals()`: calcula las normales de los vértices del modelo. Este es un procedimiento costoso y no debe llamarse si las normales ya están calculadas (por ejemplo, por que han sido definidas en el archivo .obj).

`C3DModel::SetObjectMaterial(int whichObject, CTextura* texture)`: Asigna la textura o material al objeto indicado por un índice.

Para utilizar la clase ObJMeshLoader necesitamos

```
ObjMeshLoader::Import(char*strFileName,C3DModel *model);
```

Esta función recibe como primer argumento la ruta al archive .obj que queremos cargar, y un puntero a un `C3DModel` previamente creado, es decir, inicializado estática o dinámicamente con **new.**

Pasos

Un usuario debe seguir los siguientes pasos para poder importar un objeto representado en un archivo .obj:

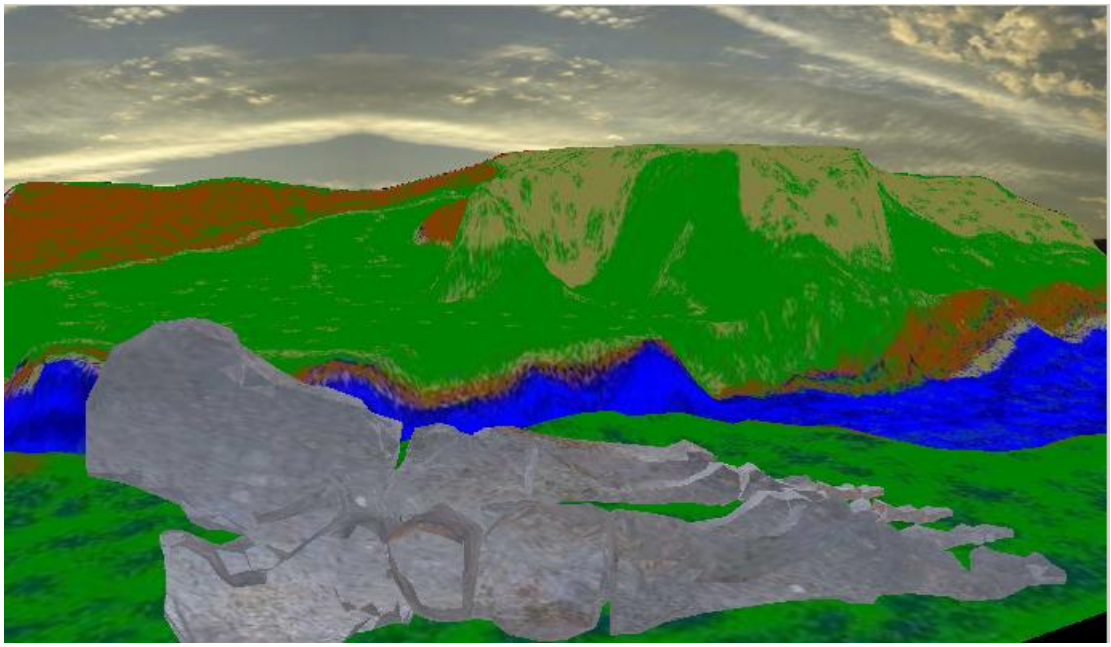
- Instanciar un objeto `ObjMeshLoader`, que se encargará de importar el modelo.
- Instanciar e inicializar un objeto `C3DModel`, que será el que contenga el objeto una vez cargado.
- Llamar a la función de importación.
- Asignar a cada objeto del modelo una textura o material, previamente creados e inicializados.

Con esto ya tenemos el objeto cargado y listo para ser usado. El modelo tendrá las mismas dimensiones y estará colocado en el mismo sitio que en el indicado por el sistema de coordenadas que el programa que lo creó, así que normalmente se hace necesario un escalado y posicionamiento previos al dibujado. Una vez hecho esto para dibujar el modelo sólo se necesita llamar al método `draw()` de éste.

Ejemplo

Con este ejemplo se puede observar la facilidad de uso y la potencia de la herramienta.

```
ObjMeshLoader loader;
CTextura *t1;
C3DModel *modelo;
...
modelo = new C3DModel;
t1 = new CTextura;
t1->cargarImagen ("data/Riverain_Valley.jpg");
t1->setFuncionPegado(GL_REPLACE);
loader.Import(modelo,"data/Ground_Valley.OBJ");
modelo->SetObjectMaterial(0,t1);
...
modelo->draw();
```



Modelos Compilados

Definición de la geometría de un modelo
3D en el código del programa

Introducción

Es interesante tener la geometría de un modelo 3D en el propio código por varias razones:

- No se necesita cargar el modelo desde el disco con un cargador específico. Implementar estos cargadores es una tarea ardua.
- Los clientes de la aplicación no tienen acceso al modelo y por tanto no lo pueden utilizar.
- Se puede acceder directamente a la lista de colores, vértices, caras…
- Se pueden crear bibliotecas de modelos para realizar demostraciones (como `glut*Teapot`)

El objetivo de este documento es crear una herramienta que transforme un archivo con la geometría del modelo a un archivo "*header*" de C con las listas de vértices, caras, normales…

Para utilizar ese modelo sólo habrá que incluir el archivo de cabecera y llamar a las funciones de renderizado, que son muy sencillas. La herramienta desarrollada se denomina "*ase2h*" y transforma archivos .ASE a un archivo .H. Los archivos .ASE son archivos que contienen la información geométrica de un modelo codificado como texto ASCII. Básicamente se trata de modificar la estructura ASCII del archivo ASE para que pueda ser leída por un compilador de C.

La desventaja de este método es que el modelo no es actualizable una vez distribuida la aplicación si no es cambiando los binarios.

Importar modelos ASE: ase2h

Un archivo ASE tiene una estructura muy definida. Cada línea empieza por un identificador que define el significado de los campos numéricos que vienen a continuación. Los identificadores tratados en esta versión de ase2h son:

Identificador	**Significado**
*MESH_NUMVERTEX	Número de vértices que posee el modelo
*MESH_FACE_LIST	Indica que a continuación empieza la lista de caras del modelo
*MESH_NUMFACES	Número de caras del modelo
*MESH_NORMALS	Indica que a continuación empieza la lista

	de normales del modelo
*MESH_VERTEX	Indica que a continuación se presenta un vértice
*MESH_FACE	Indica que a continuación se presenta una cara
*MESH_VERTEXNORMAL	Indica que a continuación se presenta una normal a un vértice determinado

Para realizar la conversión se utiliza la herramienta *gawk*, típica de los sistemas UNIX, aunque se puede encontrar una versión para Windows en http://gnuwin32.sourceforge.net/packages/gawk.htm. Esta herramienta reconoce patrones en un archivo de texto y realiza un proceso sobre las líneas del archivo. No entraremos en muchos detalles sobre la programación *awk*. El código del convertidor reconoce los patrones de la geometría del objeto y elimina la información innecesaria.

El convertidor genera 3 arrays con la información geométrica. El primero, es `vertices_ase2h[]` que es una lista de los vértices del modelo. Cada vértice es representado por 3 números float. El siguiente array generado es `caras_ase2h[]`, que contiene la información de las caras del modelo. Cada cara está compuesta por tres números enteros, por lo que las caras siempre son triangulares. Estos números enteros son índices que se deben utilizar en el array vértices_ase2h para saber a qué vértice se refiere. El último array generado es normales_ase2h[] y contiene información de la normal a cada vértice. Cada normal está compuesta por tres números float. La primera normal se refiere al primer vértice de la primera cara. La segunda al segundo vértice de la primera cara. La cuarta se referirá al primer vértice de la segunda cara y así sucesivamente.

Un mismo modelo generado en diferentes aplicaciones de diseño 3D puede generar resultados diferentes en el convertidor. La razón es que cada aplicación ha escogido un sistema de coordenadas diferente, y al mismo tiempo diferente al de OpenGL. El convertidor ase2h, en esta primera versión, trata correctamente los modelos generados por 3D Studio MAX. Esta aplicación cambia las coordenadas Z e Y (respecto al sistema de OpenGL) e invierte el sentido de Z.

Para ejecutar el programa sólo es necesario invocar al archivo .bat *ase2h* con parámetros de archivo origen y archivo destino.

El convertidor *ase2h*, el código de *gawk* y la versión más reciente de *gawk* se encuentran en el subdirectorio /convertidorASE del CD.

Dibujo del modelo del archivo .h

Una vez generado el modelo, es muy sencillo renderizarlo. Simplemente hay que iterar sobre los valores de las caras y encontrar el valor correspondiente en el array de los

vértices. Las normales se pueden referenciar en orden. El código que realiza el renderizado es el siguiente:

```
void dibuja(void)
{
        glBegin(GL_TRIANGLES);
        for(int i = 0; i < (numcaras_ase2h)*3; i+=3)
        {
                glNormal3fv(normales_ase2h + 3*i);
                glVertex3fv(vertices_ase2h + 3*caras_ase2h[i]);
                glNormal3fv(normales_ase2h + 3*(i+1));
                glVertex3fv(vertices_ase2h + 3*caras_ase2h[i+1]);
                glNormal3fv(normales_ase2h + 3*(i+2));
                glVertex3fv(vertices_ase2h + 3*caras_ase2h[i+2]);
        }
        glEnd();
}
```

Ejemplo: letras fx

El convertidor ha sido utilizado para generar la demostración dinamiclight.bpr y staticlight.bpr. Con la aplicación 3D Studio MAX se ha creado el modelo y ha sido importado con ase2h.

```
C:\convertidorAse> ase2h letrasfx.ase letrasfx.h
```

Sistemas de Partículas

Diseño de un armazón para realizar
Sistemas de Partículas en OpenGL

Introducción

La técnica más utilizada con diferencia para renderizar efectos especiales es utilizar sistemas de partículas. Un sistema de partículas consiste en una colección de elementos (partículas) que modelan un fenómeno físico. Las partículas se rigen por leyes muy sencillas, pero en conjunto simulan un comportamiento mucho más complejo.

Un buen ejemplo de fenómeno físico que se puede modelar con sistemas de partículas es una nevada. Cada copo de nieve sigue una trayectoria de caída sencilla de modelar, incluso se puede calcular como afecta el viento. Si se dibuja cada copo como un punto, el conjunto de puntos y trayectorias describe el fenómeno físico de la nieve.

El objetivo es crear una jerarquía de clases que funcionen como un armazón para crear sistemas de partículas de una forma sencilla, potente y rápida. En el diseño del armazón han influido principalmente 3 factores:

- Sencillez a la hora de diseñar Sistemas de Partículas.
- Incluir las técnicas más habituales de los Sistemas de Partículas
- Renderizado eficiente en el hardware.

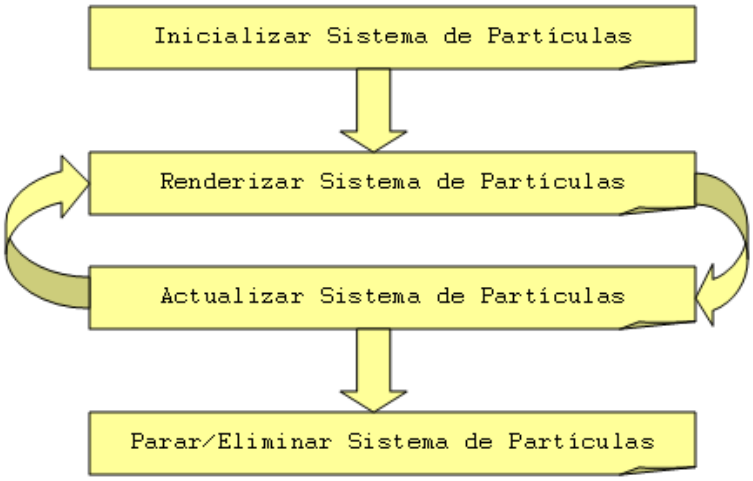
Funcionamiento del Sistema de Partículas

La idea de un sistema de partículas es muy sencilla. Se fija una tasa de emisión de partículas (número de partículas que se crean cada segundo) y se renderiza el sistema cada 1/24[(*)] de segundo para que la animación sea fluida. Cada 1/24 de segundo también se realiza la actualización del sistema. La actualización mueve las partículas en el espacio según la trayectoria individual de cada partícula. El propio Sistema lleva un contador del tiempo transcurrido entre actualizaciones para decidir cuántas partículas debe crear en cada iteración.

Añadir al código cliente un sistema de Partículas es muy sencillo. Sólo es necesario realizar 4 pasos. En el primero se crean las estructuras del sistema mediante la llamada al método inicializar del Sistema de Partículas. Este paso debe ser realizado al inicio de la aplicación, durante la inicialización de los objetos de la aplicación.

[(*)] El ojo humano por debajo de 1/24 de segundo percibe que la animación parpadea.

Durante el tiempo que dure el efecto del sistema se dibuja el sistema en el bucle de Renderizado de la aplicación. Cada cierto tiempo se actualiza el sistema para mover las partículas. Finalmente cuando el efecto de las partículas se haya terminado se para o bien se destruye el efecto.



Estructura de datos de un Sistema de Partículas extensible

Este apartado se ofrece para que el diseñador tenga una visión global de lo que sucede internamente en el Sistema de Partículas. No se mostrará código alguno ya que se ofrece el código fuente. Teniendo esta visión global se comprende esta estructura enrevesada de las partículas, pero necesaria para tener un coste muy bajo en las operaciones de eliminar/añadir/renderizar.

A simple vista, un sistema de partículas no es más que una lista de partículas. Esta lista es dinámica, se añaden y eliminan partículas continuamente. Sería un error llamar al gestor de memoria cada vez que se cree o destruya una partícula porque consumen mucho tiempo. Para solucionarlo, se utiliza un array con una longitud fija. De esta forma el sistema de partículas tiene un número de partículas máximo, pero no realiza ninguna llamada al gestor de memoria en tiempo de ejecución. Sólo resta eliminar y añadir partículas al array de forma eficiente.

Supongamos que tenemos un array con una capacidad para n partículas y que contiene actualmente m partículas en las posiciones 0..m-1. La pregunta es ¿cómo realizar las siguientes operaciones de forma eficiente?

Añadir una partícula. Si m < n basta con añadir la partícula al final del array, en la posición m. Incrementar el contador de partículas. Se consigue además que todas las partículas estén seguidas. Si m>=n no se puede añadir la partícula.

Eliminar una partícula. Supongamos que queremos eliminar la partícula p, con 0<=p<m. Mover la partícula de la posición m-1 a la posición p lo resuelve. Decrementar el contador de partículas. Además volvemos a tener todas las partículas activas seguidas.

En todas las fuentes consultadas se utiliza un array de partículas como estructura de datos básica. Una vez decidido cómo representar el sistema, queda tratar las partículas.

Parámetros que pueden influir en una partícula:

- Posición en el espacio.
- Velocidad y dirección del movimiento.
- Tiempo de vida.
- Color.
- Tamaño.
- Textura.
- Temperatura.
- Ángulo de rotación sobre si misma.
- Otros muchos...

La representación más lógica para la partícula es una estructura con todos esos campos, pero ¿todos los sistemas de partículas pueden ser creados con estos parámetros? ¿Los hemos listado todos? Podría ser que sí, pero sería más realista pensar que no. La representación de una partícula entonces puede cambiar fácilmente según el fenómeno representado.

La orientación a objetos es la herramienta que se necesita para salvar estos obstáculos. El armazón provee unas clases básicas de las que el diseñador parte. La clase básica de las partículas tendrá los parámetros que se presentan con más frecuencia. Si estos parámetros no son suficientes para el diseñador, éste derivará una nueva clase añadiendo los que necesite.

El array de las partículas por tanto contiene punteros a los distintos objetos partícula del sistema, que pueden referirse también a las clases que haya derivado el diseñador. Queda tratar quién crea esas partículas puesto que el sistema de partículas no sabe que partículas instanciar. Para ello se define un método factoría que debe implementar el diseñador, instanciando la clase Partícula necesaria.

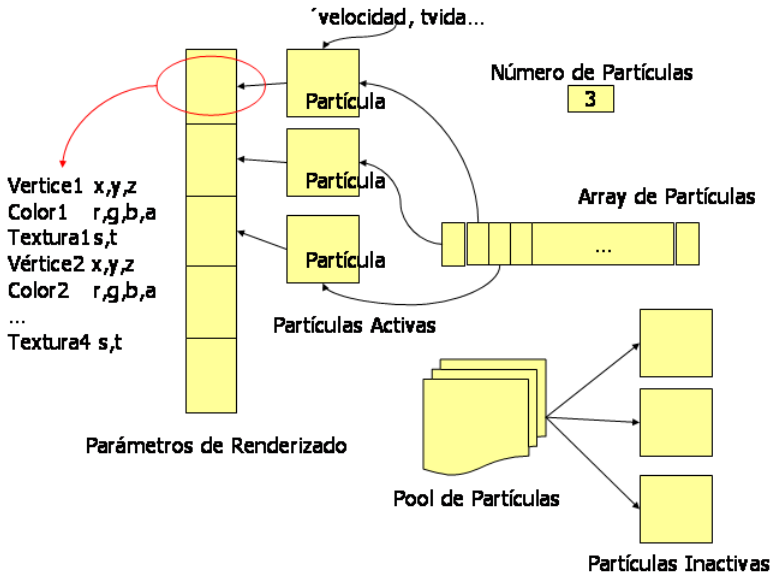
La creación de las partículas se realiza cuando se construye el objeto Sistema de Partículas. En ese momento se llama al método factoría de partículas para obtener el número máximo de partículas que puede albergar el sistema. No será necesario más adelante crear partículas, simplemente se reutilizarán esos objetos. Claro está que necesitaremos una estructura para albergar estas partículas inactivas, que se han creado y que pueden ser utilizadas más adelante. Utilizamos una pila para guardar las partículas inactivas. Cuando se necesite crear una partícula sólo tenemos que buscar en la cima de la pila. Cuando se elimine una partícula se añadirá a la cima de la pila.

Para un Renderizado eficiente se utilizan los "*Vertex Arrays*" de OpenGL. La forma tradicional de pasar los vértices al render de OpenGL es utilizar las funciones glVertex. El inconveniente de este método es que hay que realizar una llamada por cada vértice. Además hay que especificar el color, las coordenadas de textura... Los "*Vertex Arrays*" son arrays que contienen toda la información geométrica (posición, normal, textura, color) de varios vértices, de forma empaquetada. Ahora en vez de realizar una llamada por cada vértice, se realiza una llamada para todo el conjunto de vértices. Siempre es más eficiente mover un bloque de información grande que mover varios bloques de tamaño mucho más reducido. Es por esto que dividimos la información de una partícula en dos partes:

- Parámetros de Renderizado. Geometría OpenGL de la partícula.
- Parámetros de Actualización. Parámetros para la actualización de la partícula.

Todos los parámetros de Renderizado de todas las partículas se encuentran en memoria de forma continua. Los Parámetros de Actualización no hace falta que se encuentren empaquetados en memoria y son la parte "extensible" de la partícula.

La clase partícula es la clase base para las partículas derivadas por el diseñador. Contiene un puntero a los parámetros de renderizado de esa partícula, que el diseñador no es responsable de asignar. También contiene los parámetros extensibles de velocidad, tiempo de vida... El diseñador puede añadir tantos parámetros como sea necesario.



Ventajas de esta estructura:

- Parámetros de las partículas extensibles.
- Renderizado eficiente de todas las partículas.

- No realiza llamadas al gestor de memoria del SO.
- Comportamiento de las partículas extensible.

Desventajas:

- Llamadas a métodos virtuales.
- Complejidad

Tipos de Sistemas de Partículas

Se han distinguido dos clases de Sistemas de Partículas:

- ParticulaSistema: Los que renderizan sus partículas como puntos.
- ParticulaSistemaQUADTex: Los que renderizan sus partículas como Cuadrados texturados.

El diseñador tiene que decidir cuál de ellos es más apropiado para sus necesidades. El primero se debería utilizar para sistemas de partículas que dibujen puntos. Un ejemplo sería una nevada. Cada copo de nieve se puede representar como un punto de la pantalla. Los puntos pueden tener tamaños más grandes que un píxel. Son más eficientes los sistemas que utilizan puntos porque requieren menos información para dibujar cada partícula. Cada partícula consta de un vértice y de un color.

Los sistemas de partículas que necesiten de textura tendrán que utilizar los *QUADS* texturados. Para dibujar un *QUAD* texturado se necesita mucha más información. Cada partícula consta de 4 vértices, 4 colores y 4 coordenadas de textura.

Si se posee el Hardware necesario, el primer sistema de Partículas puede comportarse de la misma forma que los segundos. La extensión *GL_NV_point_sprite* permite aplicar una textura a un punto, sin necesidad de coordenadas de textura. La extensión *GL_ARB_point_parameters* hace que el tamaño de un punto pueda variar según la distancia al observador, y así conseguir la ilusión de profundidad con simples puntos.

En esta tabla se muestran las características de cada uno:

Característica	**ParticulaSistema**	**ParticulaSistemQUADTex**
Número de vértices que se tienen que actualizar en cada iteración	1	1
Puede utilizar textura	No / Si (con la extensión *GL_NV_point_sprite*)	Si

Las partículas se ven afectadas por la distancia al observador.	No / Si (con la extensión *GL_ARB_point_parameters*)	Si
Se pueden aplicar efectos de BLENDING al renderizar	No / Si (con la extensión *GL_ARB_point_parameters*)	Si
Tamaño en Bytes de los parámetros de renderizado	16 bytes	96 bytes
Necesitan coordenadas de textura	No	Si

La clase ParticulaSistema

La clase que encapsula un Sistema de Partículas se llama `ParticulaSistema`. Esta clase renderiza las partículas como puntos. También existe la clase derivada `PariculaSistemaQUADTex` que renderiza las partículas como cuadrados texturados. Las dos clases tienen los mismos métodos, sólo difieren en el comportamiento de algunos métodos, por lo que nos referiremos como `ParticulaSistema` a los dos. A continuación se listan las capacidades de la clase.

`Textura`

Todas las partículas del sistema comparten la misma textura, mediante un objeto `CTextura`. Se puede especificar al sistema que textura utilizar mediante dos funciones `setTextura(CTextura *tex)` y `setTextura(char* rutaTextura)`. La primera fija una textura creada por el cliente, y es responsable de eliminarlo. La segunda función crea una textura del archivo de imagen especificado en `rutaTextura` con las propiedades por defecto. Cada vez que se vaya a renderizar el sistema de partículas, se activa la textura y por lo tanto se aplica con las propiedades fijadas en esa textura. Si no se realiza ninguna llamada a ninguno de los dos métodos, las partículas no estarán texturadas.

`Característica`	`Valor por defecto`
Filtro Mag/Min	*POINT SAMPLING*. Es la función más rápida para aplicar una textura.
Función de pegado	*GL_MODULATE*. Para que las partículas tengan el color del polígono al que se aplica la textura.
Función de mezcla	*MEZCLA_ADITIVA*. Mezcla más habitual en los sistemas de partículas.

Tamaño de Partícula

Todas las partículas del sistema tienen el mismo tamaño y se fija mediante la función `setTamanoParticulas(float tam)`. Se puede obtener el tamaño utilizado por un sistema se particulas con el método `getTamanoParticulas(void)`.

Tasa de emisión de las partículas

La tasa de emisión es un parámetro imprescindible en cualquier sistema de partículas. En este sistema, se mide con las unidades Partículas/sg. El propio objeto del sistema mantiene un contador que mide el tiempo transcurrido entre dos llamadas al método `actualizar()` para poder emitir el número de partículas correspondientes a ese intervalo de tiempo. La tasa de emisión se fija con el método `setTasaEmisionParticulas(float tasa)` y se puede obtener con el método `getTasaEmision()`. Si en un determinado momento el sistema de partículas está en la máxima capacidad, no se crearán nuevas partículas.

Extensiones disponibles

Mediante la llamada al método `estaDisponible(int extension)` se puede obtener si está disponible una extensión. Las extensiones que se pueden consultar están en la siguiente tabla. Este método resulta útil para saber si un `ParticulaSistema` renderizará las partículas como puntos o como *QUADS*.

Identificador	Extensión
ARB_MULTITEXTURADO	Consulta la disponibilidad de la extensión *GL_ARB_multitexture*, pero no añade ninguna funcionalidad al sistema.
ARB_POINTPARAMETER	Consulta la disponibilidad de la extensión *GL_ARB_point_parameters* y si está disponible, las particulas que se dibujen como puntos ven su tamaño afectado por la distancia al observador.
NV_POINTSPRITE	Consulta la disponibilidad de la extension *GL_NV_point_sprite* y si está disponible, se puede aplicar una textura a las partículas dibujadas como puntos. No es necesario utilizar QUADS.

Emisión de partículas y reset del sistema

Se puede pedir al sistema que emita explícitamente un número de partículas, sin que tenga nada que ver con la tasa de emisión. Esto resulta útil para no empezar con un

sistema de partículas vacío. La función es `emitir(unsigned n)`. Si se quiere dejar el sistema en el estado inicial (ninguna partícula) se utiliza el método `reset(void)`.

Renderizado del sistema

Para dibujar el sistema de partículas entero sólo es necesario realizar una llamada al método `renderizar(void)`. Esta llamada no realiza ninguna actualización sobre el sistema.

El plano de rebote

Es muy usual en los sistemas de partículas utilizar un plano donde reboten las partículas, pero sólo uno para no realizar muchos cálculos. La clase partícula permite asignar un plano de rebote en cada sistema de partículas con el método `setPlanoRebote(float A, float B,float C, float D)`, que simboliza el plano con la ecuación Ax + By + Cz + D = 0. Una vez establecido, se puede utilizar el método `interseccion(…)` para obtener todos los parámetros del choque de una partícula con el plano designado. La llamada a `intersección(…)` se realiza por el cliente, ya que es el responsable de actualizar las trayectorias de las partículas. El proyecto `llama.bpr` que aparece en el CD que se adjunta, contiene un ejemplo de cómo tratar un rebote de partículas.

Obtener el número de partículas

El método `getNumParticulasEmitidas()` obtiene el número de partículas emitidas desde la inicialización del objeto o desde el último reset. El método `getNumeroParticulas()` obtiene el número de partículas activas en el instante de la llamada. El método `getTamMax()` obtiene el número máximo de partículas que pueden estar activas a la vez en el sistema.

Emisores

Consultar el apartado siguiente para una explicación detallada de los emisores. Se enlaza un determinado emisor con un sistema de partículas con el método `setEmisor(ParticulaEmisor* emisor)` y se obtiene el emisor activo de un sistema con el método `getEmisor()`.

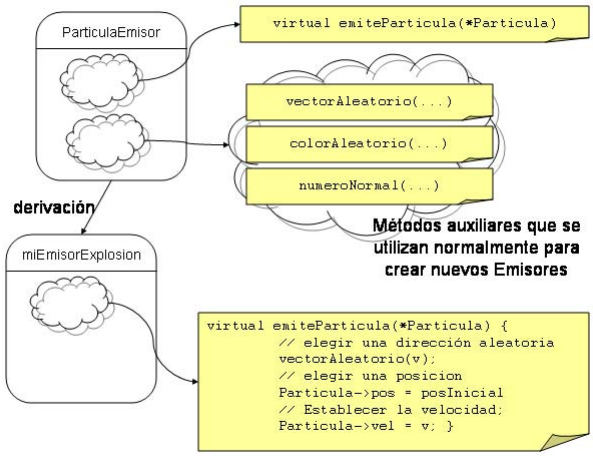
Actualizar el Sistema

Para poder obtener una animación con el sistema de partículas se debe actualizar el sistema cada cierto tiempo con la llamada al método `actualizar()`. Este método actualiza las posiciones de las partículas (según lo especificado en el código cliente) y emite las partículas correspondientes.

Emisores de Partículas

Todos los sistemas de partículas tienen algo en común: hay un objeto que emite las partículas. En una nevada el cielo emite las partículas con velocidad hacia el suelo y con posición en el cielo. En una explosión, el centro de la explosión emite las partículas hacia todas direcciones.

Hemos querido abstraer este conocimiento en una clase. Como varios sistemas de partículas pueden compartir un mismo emisor el diseñador se ahorra tiempo de desarrollo. Si necesita un emisor especial, puede derivar una nueva clase. La clase base de los emisores `ParticulaEmisor` ofrece una serie de funciones auxiliares que ayudan en la tarea de definir un nuevo emisor. Además el armazón ofrece 4 tipos de emisores ya implementados.



Cuando se quiere crear un nuevo sistema de partículas se debe especificar que emisor "emite" las partículas. El sistema de partículas lo utiliza internamente para crear las partículas.

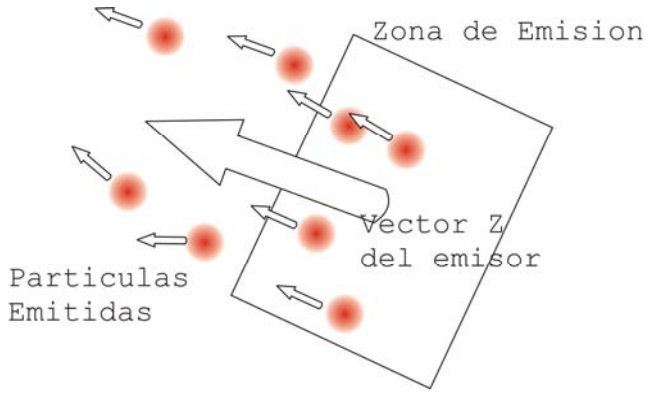
Para crear un nuevo emisor personalizado sólo es necesario implementar la función `emiteParticula(Particula* P)`, que inicializa los campos posición y velocidad de la partícula. El concepto de emisor se adapta muy bien a una jerarquía de clases. El objetivo principal de los emisores es la movilidad del sistema de partículas. Cambiando de lugar al emisor o rotándolo, las partículas cambian su punto de emisión o dirección de emisión. Es por ello que los emisores tienen un sistema de coordenadas, que puede ser modificado mediante 4 funciones.

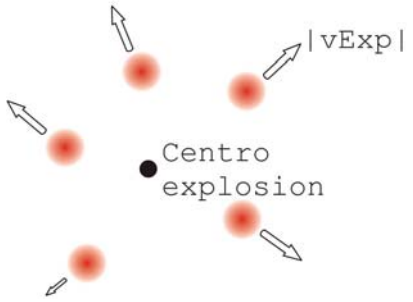
Funcion	Significado
`rotateGlob(float deg, float`	Realiza una rotación de `deg` grados del

`x, float y, float z)`	eje de coordenadas del emisor respecto de las coordenadas absolutas.
`rotateLoc(float deg, float x, float y, float z)`	Realiza una rotación de `deg` grados respecto al sistema de referencia del emisor.
`moveGlob(float x, float y, float z, float distance)`	Realiza una traslación del eje de coordenadas del emisor respecto de las coordenadas absolutas.
`moveLoc(float x, float y, float z, float distance)`	Realiza una traslación del eje de coordenadas del emisor respecto al sistema de referencia del emisor

Inicialmente, un emisor se encuentra en el origen de coordenadas (0,0,0) y con los ejes de coordenadas coincidentes con los de OpenGL. También se obtienen ventajas reutilizando los emisores para distintos sistemas de partículas. En esta primera versión se han implementado dos tipos de emisores para los dos tipos de Sistemas de Partículas:

- ParticulaEmisorRectangular. Emite partículas en un rectángulo del plano XY del sistema de coordenadas del emisor. Las partículas son emitidas en la dirección del vector Z. Es una clase abstracta. Las clases utilizables son `ParticulaEmisorRectangularPoint` y `ParticulaEmisorRectangularQUADTex`
- ParticulaEmisorExplosion. Emite partículas en la posición del emisor en todas direcciones. Es una clase abstracta. Las clases utilizables son `ParticulaEmisorExplosionPoint` y `ParticulaEmisorExplosionQUADTex`.





Clase de emisor	Características
`ParticulaEmisorRectangularPoint`	Emite partículas con parámetros de renderizado de punto. Las partículas se posicionan en un rectángulo con centro en el sistema de referencia del emisor y sobre el plano XY. Las partículas tienen la dirección del eje Z del eje del emisor.
`ParticulaEmisorRectangularQUADTex`	Emite partículas con parámetros de renderizado de *QUAD* texturado. Las partículas se posicionan en un rectángulo con centro en el sistema de referencia del emisor y sobre el plano XY. Las partículas tienen la dirección del eje Z del eje del emisor.
`ParticulaEmisorExplosionPoint`	Emite partículas con parámetros de renderizado de punto. Las partículas se posicionan en el centro del sistema de referencia del emisor. Las partículas tienen la dirección del eje Z del eje del emisor.
`ParticulaEmisorExplosionQUADTex`	Emite partículas con parámetros de renderizado de *QUAD* texturado. Las partículas se posicionan en el centro del sistema de referencia del emisor. Las partículas tienen la dirección del eje Z del eje del emisor.

Aleatoriedad en la emisión

La clave para la apariencia de un sistema de partículas es la aleatoriedad. Si todas las partículas tienen la misma velocidad, dirección o punto de emisión, el sistema posee una uniformidad exagerada, y no es visualmente atractivo. La clases implementadas de emisores puede emitir partículas según una función de probabilidad. Por ejemplo, se

puede fijar que la velocidad de las partículas emitidas siga una normal de media v con desviación típica σ.

Función de clases derivadas de ParticulaEmisorRectangular	**Significado**
`setZonaEmision(float ladox, float ladoy)`	Establece la zona de emisión en un rectángulo centrado en la posición del emisor. Reside en el plano XY del emisor y tiene unas dimensiones de *2*ladox* en el eje X y *2*ladoy* en el eje Y. Las partículas se distribuyen de manera uniforme en la superficie.
`setVarianzaDireccion(float sigma)`	Establece la desviación típica que se utiliza para decidir la dirección de la partícula para cada una de las componentes x,y,z. La media es la dirección Z del eje del emisor. Con *sigma* igual a cero, todas las partículas tienen la misma dirección.
`setVarianzaDireccion(float sigmax, float sigmay,float sigmaz)`	Establece la desviación típica que se utiliza para decidir la dirección de la partícula para cada una de las componentes x,y,z por separado. La media es la dirección Z del eje del emisor. Con las tres *sigma* igual a cero, todas las partículas tienen la misma dirección.
`setVelocidad(float vel, float sigmaVel)`	Establece el módulo del vector de dirección, con la desviación típica asociada. Si *sigma* es igual a cero, todas las partículas tienen la misma velocidad.
`float* getDireccion()`	Obtiene el vector Z del sistema de coordenadas del emisor.
`getVelocidad(void)`	Obtiene el módulo de la velocidad del emisor.

Función de clases derivadas de ParticulaEmisorExplosion	**Significado**
`setVelocidad(float vExp)`	Fija el módulo de la velocidad con la que serán emitidas las partículas.
`float getVelocidad(void)`	Obtiene el módulo de la velocidad

Funciones auxiliares estáticas de ParticulaEmisor	**Significado**
`vectorAleatorio(float* v)`	Crea un vector aleatorio de módulo 1 en cualquier dirección.
`vectorAleatorioNormal(float *dir,float sigma, float* res)`	Crea un vector aleatorio de módulo 1 en *res* que sigue a una normal de desviación típica *sigma* y media el vector *dir*.
`colorAleatorio(unsigned char* color)`	Crea un color aleatorio de 4 unsigned bytes
`numeroNormal(float media, float sigma)`	Valor de una normal(*media*, *sigma*)
`rellenarColor( ParamRenderQuadGL_T2F_C4UB_V3F* estr, unsigned char* color)`	Rellena los campos de color de la estructura *estr* con el color indicado *color*.
`rellenarColor( ParamRenderPointGL_C4UB_V3F* estr, unsigned char* color)`	Rellena los campos de color de la estructura *estr* con el color indicado *color*.
`rellenaTextura( ParamRenderQuadGL_T2F_C4UB_V3F* estr)`	Rellena los campos de la estructura *estr* de las coordenadas de textura por defecto para un *QUAD* texturado.

Diseño de Sistemas de Partículas Personalizados

Los sistemas de partículas personalizados se obtienen derivando de la clase abstracta `ParticulaSistema` o de `ParticulaSistemQUADTex`. Entonces se implementan los métodos virtuales que se ocupan del comportamiento personalizado de las partículas. También será necesario asignar un emisor, bien de los que ofrece el armazón o bien uno diseñado especialmente.

Hay cuatro funciones que hay que implementar:

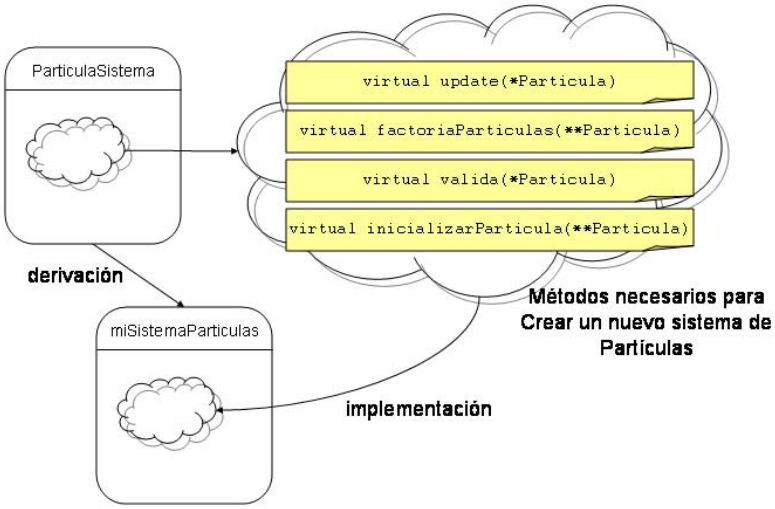
- `update(Particula* P):` Recibe como parámetro una partícula cuyos parámetros deben ser actualizados para la siguiente iteración. Por ejemplo, para un sistema de Partículas de nieve, hay que avanzar la posición de la partícula en la dirección que indica la velocidad (hacia abajo). En caso de actualizar una partícula con parámetros de renderizado de QUADTex, no es necesario actualizar los cuatro vértices. Sólo el primero es necesario ya que el sistema de Partículas rellena los otros 3 para ofrecer *billboarding*.

- `factoriaParticulas(Particula** P):` Este es el método que instancia las partículas. Si no se ha derivado ninguna clase nueva de Partícula simplemente habrá que instanciar un objeto Partícula y devolverlo. Si se ha derivado una nueva clase, habrá que instanciar un objeto de esa clase.

- `valida(Particula* P)`: Este método informa al sistema de partículas cuando una partícula debe ser eliminada. En cada iteración se realiza una llamada a esta función para consultar el estado de la partícula. Normalmente las partículas se eliminan cuando llevan un tiempo de vida determinado o cuando llegan a una determinada posición (cuando llegan al suelo en el caso de una nevada).

- `inicializarParticula(Particula* P)`: Este método inicializa los parámetros de la partícula que han sido añadidos por el sistema derivado. También se puede utilizar para inicializar los campos que no ha tratado el emisor.

Una vez creada la clase del sistema de

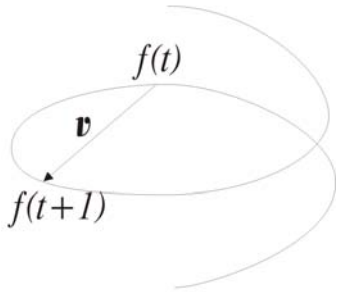


partículas sólo queda ajustar los parámetros del sistema referenciados en el apartado La Clase `ParticulaSistema`.

Ejemplo: Nieve

En este ejemplo se pretende simular una nevada mediante el dibujo de cada copo de nieve como un punto. Los copos de nieve se crearán a una cierta altura y desaparecerán cuando lleguen al suelo. Para hacer más realista la animación, los copos de nieve no caerán en línea recta, si no que describirán una trayectoria en hélice. Las partículas también pueden ver afectado su movimiento por el efecto del viento.

Para la actualización de la trayectoria del copo se necesitan algunos parámetros más. Creamos la clase `ParticulaCopo` derivada de `Particula`, añadiendo los campos `t` y `r`. El parámetro `t` es utilizado para actualizar la trayectoria paramétrica de la partícula. El parámetro `r` mide el radio de la espiral de caída.



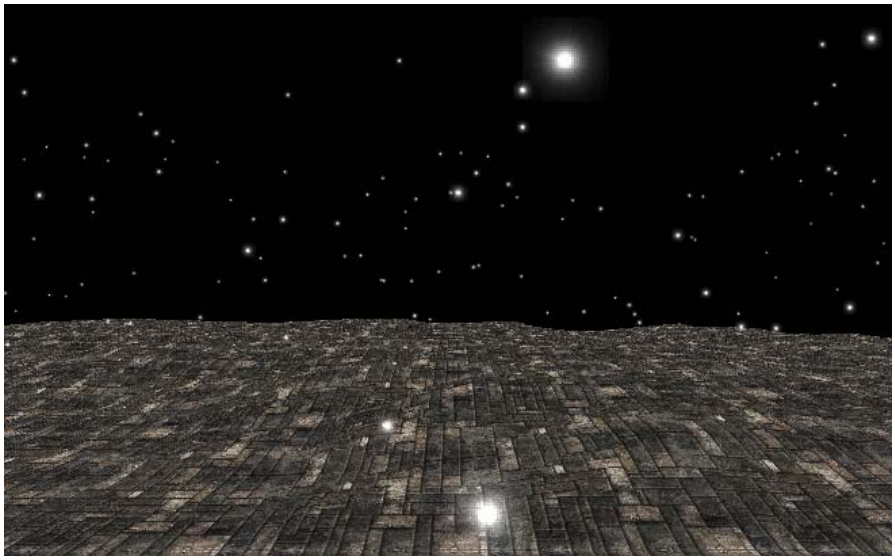
Trayectoria
del copo

Ahora elegimos el sistema de Partículas del cual derivar: `ParticulaSistema`. Las partículas serán consideradas como puntos. Derivamos la clase `ParticulaSistemaNieve` de `ParticulaSistema` e implementamos las funciones virtuales `update`, `factoriaParticulas`, `valida` e `inicializarParticula`.

El emisor debe ser un rectángulo a una determinada altura. Para eso situamos el sistema de coordenadas de `ParticulaEmisorRectangularPoint` a una altura h. También se debe rotar el emisor para que el eje Z apunte hacia abajo. Con la llamada al método `setZonaEmision(float ladox, float ladoy)` se establece la zona de emisión como un rectángulo centrado en el eje de coordenadas del emisor. Las dimensiones son *`2*ladox*2*ladoy`*. Al final, el emisor utilizado emite las partículas en un rectángulo con una velocidad negativa en el eje Y absoluto. Este emisor podría ser reutilizado por ejemplo para los efectos metereológicos.

Las constantes del código son resultado de la experimentación. Deben variarse según el volumen de vista definido en OpenGL para obtener un efecto realista.

Una vez creadas estas clases, sólo hay que añadir al código cliente la Inicialización, el Renderizado y la Actualización. El resultado final se encuentra en el proyecto de C++ Builder nieve.bpr (incluido en CD que se adjunta). Se ofrece todo el código fuente. La demostración permite que se cambie la tasa de emisión y la velocidad de las partículas en tiempo de ejecución, con las teclas m, n y +, - respectivamente. Además con la tecla i se muestra información sobre los parámetros del sistema de partículas y la posición y orientación del eje de coordenadas del emisor.



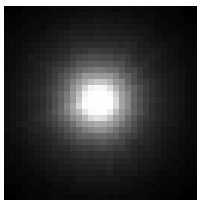
Nota: La visualización puede cambiar según el hardware gráfico por la disponibilidad de las extensiones `NV_point_sprite` y `ARB_point_parameters`.

Ejemplo: Explosión

A simple vista una explosión no parece un fenómeno sencillo de simular. Para simular estos efectos es necesario abstraerse y pensar en términos de partículas, es decir, que comportamiento tendría un punto en una explosión. Las características de este sistema de partículas son:

- Todas las partículas se emiten en el mismo punto, en el centro de la explosión. Para cada partícula se elige una dirección aleatoria.
- A medida que las partículas se alejan del centro pierden energía. Al perder energía se dibujan con colores más apagados hasta desaparecer.
- Las partículas se emiten a la vez.

No hace falta derivar una nueva clase para las partículas ya que tenemos todos los parámetros necesarios en la clase `Particula`. Dibujaremos todas las partículas con esta textura:



Si se aplica esta textura a un *QUAD* que tiene un color determinado y seleccionamos la función *GL_MODULATE* al aplicar la textura, el círculo aparecerá coloreado. Activando la mezcla *GL_BLEND* los bordes coloreados en negro desaparecerán. Para

conseguir que las partículas vayan desapareciendo a medida que se alejan del centro se disminuye el canal alpha del *QUAD*. Cuando una partícula tiene el canal alpha a 0, no se dibuja nada.

Para conseguir el efecto de explosión, se fija una tasa de emisión muy alta comparada con el número de partículas. Así se emitirán todas las partículas a la vez y no se podrán emitir más porque se ha llegado al límite.

La temperatura de cada partícula está simbolizada por el campo `tvida`. Cuanto más bajo sea, más temperatura tiene la partícula y debería ser pintada con un color acorde a la temperatura. Una técnica muy extendida es utilizar una paleta de colores. Antes de crear el sistema de partículas se crea una degradación de color que contiene los colores por los que debe pasar una partícula del sistema. Así, utilizando como índice de la paleta el campo `tvida` de la partícula, se consigue simular el color real de la partícula. Para dar soporte a la multitud de Sistemas de Partículas que necesitan una Paleta de colores , se ha añadido al armazón de los sistemas de partículas la clase `CPaleta`.

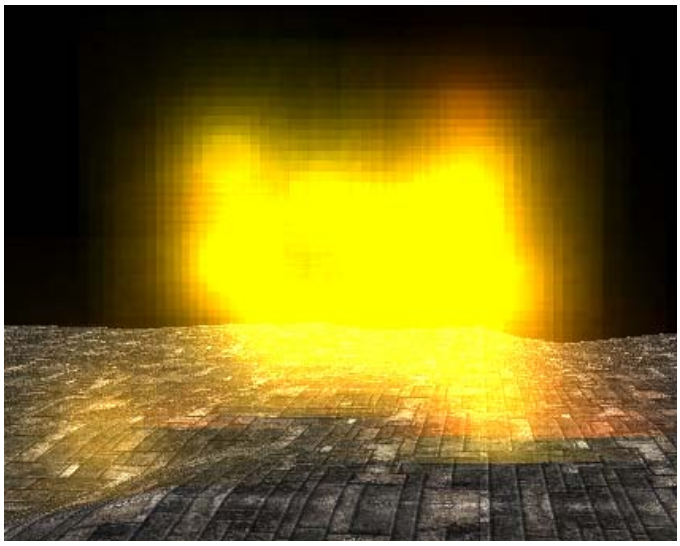
La degradación de color se crea con colores clave. Una paleta almacena 256 colores en orden. Al fijar una posición de la paleta con un color, se crea un color clave en esa posición. Cuando se han fijado los colores clave de la paleta, se realiza una degradación de color en las posiciones de la paleta que no son colores clave, consiguiendo una degradación suave entre varios colores clave. Al crear un objeto `CPaleta`, la degradación por defecto es una transición entre el color blanco (color clave de la posición cero) y el color negro (color clave de la posición 255), una escala de grises.

Funciones	**Significado**
`setColor(int key, unsigned char rojo, unsigned char verde, unsigned char azul)`	Asigna el color clave (`rojo`, `verde`, `azul`) en la posición `key` de la paleta.
`getColor(int key, unsigned char *rojo, unsigned char *verde, unsigned char *azul)`	Obtiene el color (`rojo`, `verde`, `azul`) de la posición `key` de la paleta.
`getColor(int key, unsigned char *color)`	Obtiene el color de la posición `key` de la paleta, almacenándolo en el vector `color`.
`render(float tamX, float tamY)`	Renderiza los colores de la paleta en un rectángulo de tamaño `tamX*tamY`.

El emisor utilizado es `ParticulaEmisorExplosionQUADTex`, y sólo hace falta posicionarlo en el lugar de la explosión. La velocidad de la explosión se fija con el método `setVelocidad(float vel)`.

El resultado final se encuentra en el proyecto de C++ Builder explosion.bpr (incluido en el CD). Se ofrece todo el código fuente. Pulsando la tecla i se muestra información

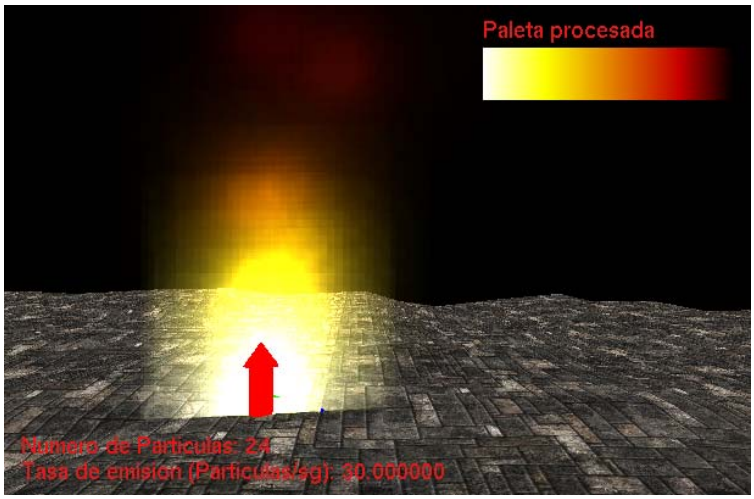
sobre los parámetros del sistema de partículas y la posición y orientación del eje de coordenadas del emisor. Además renderiza la paleta utilizada por el sistema.



Ejemplo: Fuego

La simulación del fuego es muy parecida a la de la explosión. También necesita una paleta de color para simular el color de las partículas. Lo único que cambia es la emisión de las partículas. La emisión se realiza en un punto con un `ParticulaEmisorRectangular` en dirección al eje positivo Y. Es muy importante introducir variaciones en la dirección y velocidad de las partículas para obtener la naturaleza caótica del fuego. La textura utilizada es la misma que en el caso anterior.

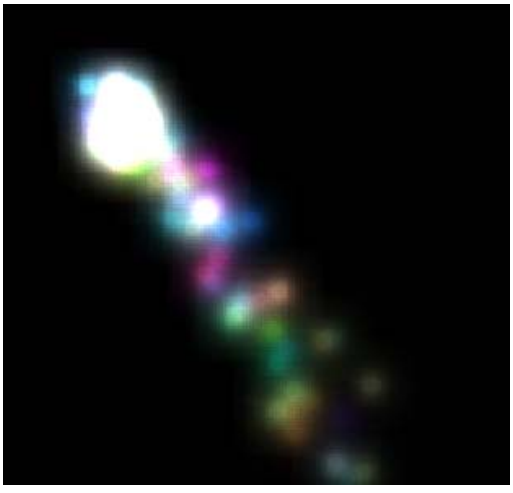
El resultado final se encuentra en el proyecto de C++ Builder fuego2.bpr (incluido en el CD). Se ofrece todo el código fuente. Pulsando la tecla i se muestra información sobre los parámetros del sistema de partículas y la posición y orientación del eje de coordenadas del emisor. Además renderiza la paleta utilizada por el sistema.



Otros Ejemplos

Se ofrece el código fuente de otros tres ejemplos más de sistemas de partículas:

- Movimiento. Ofrece una visión sobre las posibilidades de movimiento de los emisores.
- Llama. Ejemplo que muestra cómo utilizar la característica del rebote.
- Humo. Simula una columna de humo.



Otros tipos de representación de las partículas

El diseñador puede decidir que sus partículas no son adecuadas para ser dibujadas como puntos o como QUADS. También puede querer implementar su propia

función de renderizado. Para permitir esta extensión se han creado los métodos virtuales `creaArrayParametrosRender(char** dest, unsigned n)` y `sizeofParametroRender(void).` Gracias a estos métodos se pueden utilizar distintos tipos de parámetros de renderizado. Sólo es necesario que el cliente cree en memoria la estructura y que indique el tamaño en bytes de los parámetros de renderizado de la partícula. Por último deberá implementar la función `renderizar()` que sepa como tratar estos nuevos parámetros de renderizado. Esta extensión necesita bastante conocimiento de la estructura interna del sistema de partículas, pero por lo menos no imposibilita tal extensión. Con este método se ha creado la clase `ParticulaSistemaQUADTex`, derivada de `ParticulaSistema`.

Conclusión

Este armazón de sistemas de partículas es capaz de implementar un número elevado de sistemas de partículas gracias a las numerosas herramientas que posee y su capacidad de extensión. Se pueden añadir ciertas características que aun lo harían más atractivo:

- Interacción entre partículas.
- Varios planos de rebote.
- Métodos para resolver ecuaciones diferenciales en el propio sistema, que no sean implementados por el diseñador.
- Trayectorias predefinidas para los emisores.
- Trayectorias predefinidas para las partículas.

Efecto: Sombras

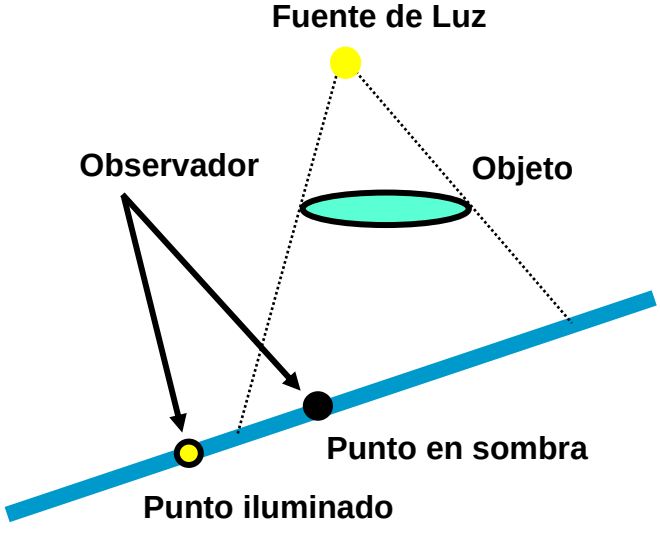
Introducción

La utilización de sombras en una escena 3D otorga un gran realismo. El problema surge cuando se utilizan múltiples puntos de luz o hay una gran densidad de objetos en la escena. Hay varias formas de generar sombras para un objeto. La tradicional es la *proyección de los vértices* sobre el plano en que se quiere ver la sombra. Esto no es siempre posible, sobre todo para objetos muy complejos o para planos de proyección no uniformes.

Explicación del Efecto

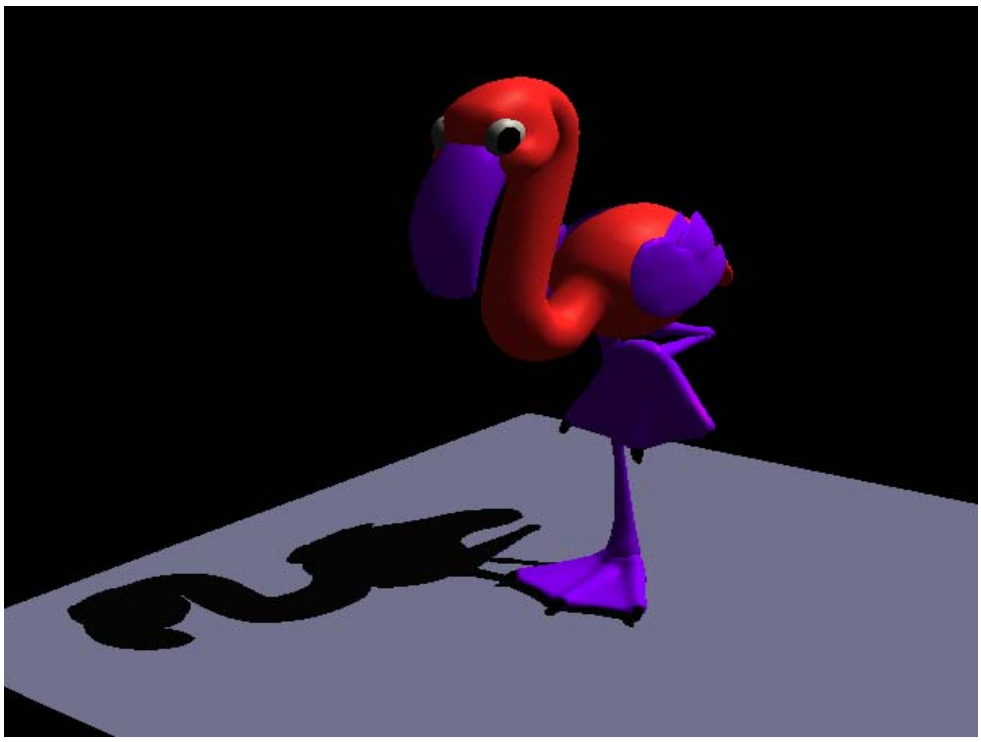
El campo de las sombras en OpenGL es muy amplio, debido a que el lenguaje en sí no las implementa, y que los programadores tienen que implementar sus propios algoritmos de sombreado en sus escenas. Estos algoritmos tienen su representación más sencilla en el caso de las sombras proyectadas, y se van complicando a cada añadido que se hace.

Empecemos primero por las ideas básicas de las sombras. Las sombras surgen en las escenas con la aparición de las luces. Una luz genera una *zona de iluminación*, que puede variar según las características de la luz, del medio en el que se encuentre, etc. Una sombra es una obstrucción en esa *zona de iluminación* por un objeto no transparente (en la realidad, todos los objetos son generadores de sombras, ya que no hay objetos "*perfectamente transparentes*", la única diferencia es que los objetos más transparentes generan sombras más suaves y menos visibles).



Lo que se pretende con la geometría proyectada, es proyectar el objeto que generaría la sombra sobre el *plano* sobre el que se pretende que se vea reflejada. Un ejemplo del objetivo de este método sería proyectar la sombra de una persona sobre el

plano del suelo, o proyectar la sombra de un vaso sobre una mesa. En la siguiente imagen tenemos una demostración de una proyección geométrica del objeto sobre el plano del suelo.



Este método se utiliza, evidentemente, cuando la sombra se proyecta sobre polígonos PLANOS, como podrían ser las paredes, el suelo, una mesa, etc.

El algoritmo básico que hay que seguir para la proyección de sombras sería el siguiente:

(1) Para cada objeto de la escena (cada fuente de sombra) tendremos que hacer lo siguiente:

   i. Calcular la matriz de proyección, dadas la posición de la luz, y el plano sobre el que se va a proyectar la misma.
   ii. Multiplicar esta matriz de proyección por la matriz de modelado actual.
   iii. Establecer el color negro, y cierto grado de transparencia (normalmente será un $\alpha = 0.5$).
   iv. Dibujar el objeto.

(2) *Dibujar la escena sin sombras*: Esta escena tendrá todos los objetos (que proyectan las sombras) y la iluminación activada (para que haya una "razón" para la existencia de las sombras).

Evidentemente, parte de este algoritmo se puede "*reducir*". Por ejemplo, si tenemos una escena con varias personas, y pretendemos proyectar sus sombras sobre el

suelo de la escena, no tendremos que calcular la matriz de proyección y multiplicarla por la matriz de modelado con cada una de las sombras. Lo que hay que hacer, es dividir la escena en "planos de proyección". Esto quiere decir que el código anterior se utilizaría para dibujar todos los objetos que proyecten sombras sobre un mismo plano de proyección. El nuevo algoritmo (más cómodo y eficiente), sería el siguiente:

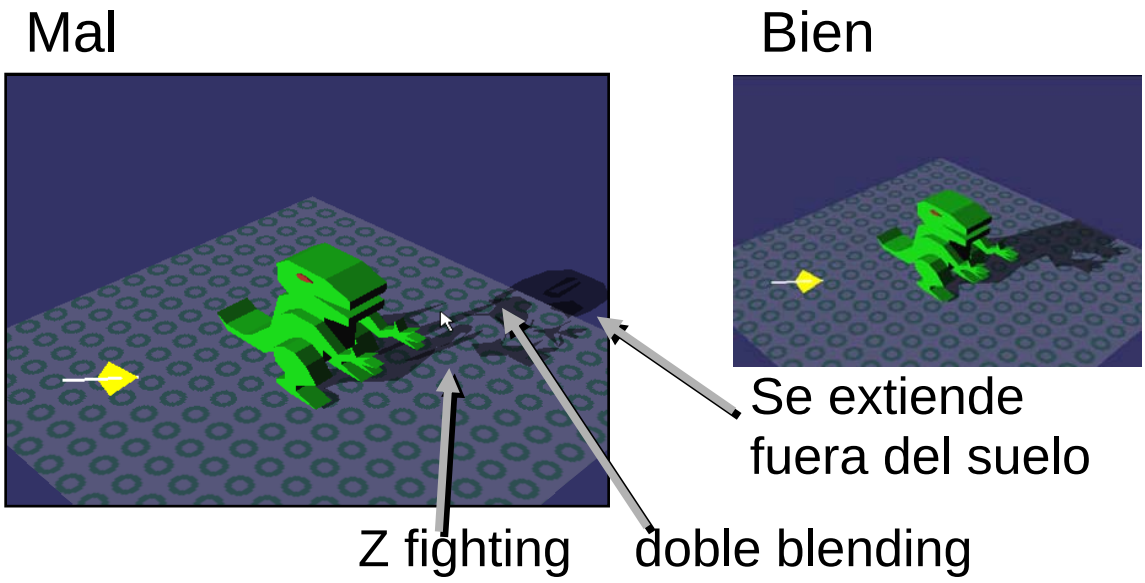
(1) Para cada plano de proyección (plano sobre el que se esperen sombras) tendremos que hacer lo siguiente:

   (a) Calcular la matriz de proyección, dadas la posición de la luz, y el plano sobre el que se va a proyectar la misma.
   (b) Multiplicar esta matriz de proyección por la matriz de modelado actual.
   (c) Establecer el color negro, y cierto grado de transparencia (normalmente será un $\alpha = 0.5$).
   (d) Dibujar cada uno de los objetos que proyectan sombras sobre este plano.

(2) *Dibujar la escena sin sombras*: Esta escena tendrá todos los objetos (que proyectan las sombras) y la iluminación activada (para que haya una "razón" para la existencia de las sombras).

Como se puede observar, la proyección de sombras es algo muy fácil, ya que se reduce a dos meras operaciones matemáticas, como son el cálculo de una matriz de proyección, y el producto de esta matriz de proyección por la matriz de modelado.

De todas formas, como cabría esperar, no es oro todo lo que reluce, y este método tiene algunos problemas. Uno de estos problemas recibe el nombre de "Z-Fighting", y surge cuando "colisionan" los objetos a dibujarse sobre el plano. Además de esto, con la proyección de sombras sin ningún añadido, las sombras se proyectan incluso fuera de la superficie sobre la que se deberían proyectar. A esto se le añade que algunos objetos, al ser proyectados, tienen partes que reciben un "doble blending". La siguiente imagen muestra los posibles problemas:



Para resolver estos problemas, se utiliza una técnica, basada en un buffer de OpenGL, que se llama *buffer de estarcido* (*stencil buffer*). El *buffer de estarcido* se usa para hacer un marcado por píxeles. Esto quiere decir, que se marcan los píxeles que se quieren dibujar, y se eliminan los otros. De esta forma, podemos marcar que sólo se dibujen las sombras en el suelo, y que se eliminen las demás. Además de esto, se consigue la eliminación del Z-fighting, y del doble blending por medio de esta técnica.

Se podría pensar que el uso del *buffer de estarcido* será costoso, ya que se tiene que mover más información sobre los píxeles a dibujar. En realidad, con las tarjetas aceleradoras de hoy en día, con memoria de 32 bits, con un *buffer de profundidad* de 24 bits, el uso de un *buffer de estarcido* de 8 no tiene costes, ya que se cargan ambos en una misma palabra de memoria. Así pues, si se está realizando un test de profundidad, el uso de este buffer no tiene costes extra.

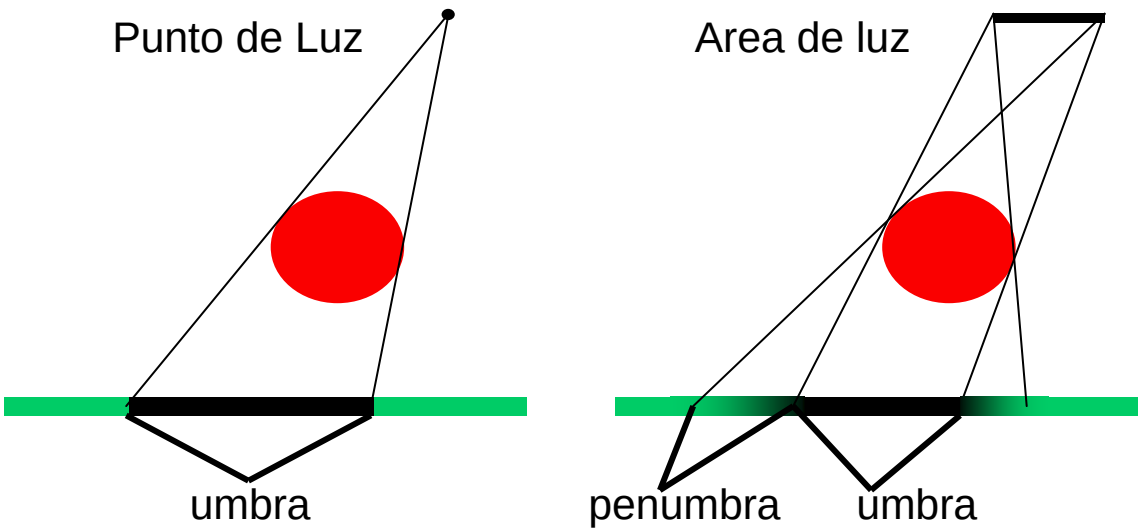
El *buffer de estarcido* tiene muchos usos. Uno de ellos, obviamente, es el de las sombras, ya que permite eliminar las partes de la proyección que se salen de nuestra superficie de sombreado. Además de eso, se puede utilizar para realizar efectos de reflejos sobre una superficie, o algunos otros efectos gráficos, basados en el marcado de píxeles (cálculo de volúmenes de sombras, por ejemplo).

Ahora explicaremos básicamente como funciona el efecto de sombras. Lo primero que tenemos que hacer, es marcar las superficies sobre las que se van a dibujar las sombras en el *buffer de estarcido*. Para hacer esto, primero desactivamos las luces, la escritura en el buffer de profundidad y en el de color (así marcaremos el suelo sin llegar a dibujarlo). Después de esto, activamos el test sobre el *buffer de estarcido*, y seleccionamos la operación a realizar (con la función *glStencilOp*). Una vez que tenemos configurado el funcionamiento del buffer, dibujamos la superficie a marcar (por ejemplo, el suelo, o las paredes de una habitación). Una vez que está marcado, reactivamos las luces y la escritura sobre los buffers de profundidad y de color, y

cambiamos la función del buffer, para que compruebe los valores. Según la función de marcado que hayamos elegido (por ejemplo, poner a 1 los píxeles del suelo), la función de comprobación que tendremos que configurar (siguiendo el ejemplo anterior, comprobar que el valor del píxel en el *buffer de estarcido* sea igual a '1'). Una vez que se activa esta comprobación, ya sólo hay que dibujar el objeto, y éste se restringirá a las superficies marcadas.

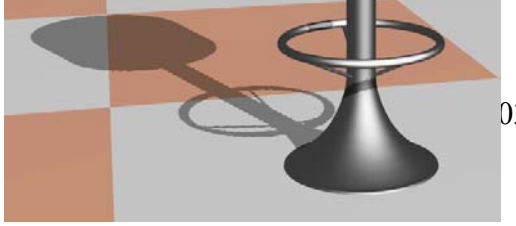
Una de las cosas que se pueden notar en las sombras proyectadas (ya estén limitadas a una superficie dada por medio del *buffer de estarcido* o no), es que la sombra está perfectamente delineada, con curvas muy definidas. En la realidad, esto no es lo que ocurre. Las sombras tienen una zona de sombreado claro, y tienen una sombra de penumbra. Este tipo de sombras, reciben el nombre de *sombras suaves* (*soft shadows*), mientras que las sombras que hemos obtenido hasta ahora reciben el nombre de *sombras duras* (*hard shadows*).

La idea que hay tras las sombras suaves, es no considerar el foco de luz como un punto fijo en el espacio, si no más bien como una "*zona de iluminación*". De esta forma, se obtienen varias sombras, muy próximas entre ellas.

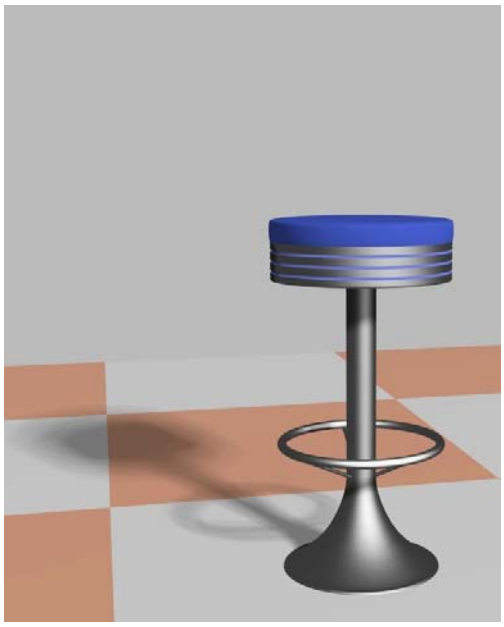


Este efecto se puede implementar de varias formas. Uno de los métodos, es usando el *buffer de acumulación*, y asignándoles diferentes grados de transparencia a cada sombra, a medida que nos alejamos del "núcleo" de la sombra (la zona en *umbra*).

En las siguientes imágenes tenemos un ejemplo de la diferencia entre un tipo de sombras y otro:



03



Sombra dura sencilla

Sombra suave, usando el *buffer de acumulación*

La diferencia en el efecto es obvia. El inconveniente que tienen las sombras suaves es obvio. Al tener que dibujar el objeto varias veces, se obtiene un coste mucho mayor. Este coste (lógicamente) varía proporcionalmente con el número de muestras que tomemos de la sombra.

Archivos

Este efecto está implementado en una clase que tiene como nombre *Sombreador*. Esta clase implementa las funciones necesarias para un marcado básico de superficies sobre las que dibujar, proyecciones, reflejos, sombras duras y suaves, etc. Además de esto, se ofrece la posibilidad de hacer que las sombras estén coloridas (en rojo). Esta opción se ofrece para poder comprobar los efectos de marcado usando el *buffer de estarcido*, ya que si el fondo es de color negro, no se puede saber si la sombra se proyecta fuera de la superficie limitada o no. Esta clase se encuentra definida en el archivo “sombras.h”.

Funciones

`Sombreador()`: Constructora de la clase sin parámetros. Esta constructora define por defecto las sombras de tipo *duras*. Además de eso, desactiva el modo de uso del *buffer de estarcido*, y de las sombras coloridas desactivadas.

`Sombreador(GLenum modo)`: Constructora del objeto *Sombreador*, en la que se define el tipo de sombras que se quieren proyectar. El valor del parámetro *modo* puede ser cualquiera de los siguientes:

- `SOMBRAS_DURAS`
- `SOMBRAS_SUAVES`

`GLenum getModo()`: Devuelve el tipo de sombras seleccionado. El valor del parámetro *modo* puede ser cualquiera de los siguientes:

- `SOMBRAS_DURAS`
- `SOMBRAS_SUAVES`

`void setModo(GLenum modo)`: Selecciona el tipo de sombras a proyectar. El parámetro *modo* puede tomar cualquiera de los valores que devuelve la función anterior.

`void activar(float groundPlane[4], float lightPos[4])`: Activa el sombreado con el modo ya seleccionado (ya sea durante la construcción del *Sombreador* o mediante la llamada a *setModo*). Los parámetros que recibe son la ecuación del plano sobre el que se va a proyectar la sombra, y la posición de la luz. Es importante tener en cuenta que en caso de que el modo activo de sombreado sea `SOMBRAS_SUAVES`, la posición de la luz se verá modificada.

`void activar(float groundPoints[3][3], float lightPos[4])`: Activa el sombreado con el modo ya seleccionado (ya sea durante la construcción del *Sombreador* o mediante la llamada a *setModo*). Recibe como parámetros tres puntos del plano sobre el que se va a proyectar la sombra y la posición de la luz. Es importante tener en cuenta que en caso de que el modo activo de sombreado sea `SOMBRAS_SUAVES`, la posición de la luz se verá modificada.

`void activar(GLenum modo, float groundPlane[4], float lightPos[4])`: Activa el sombreado con el modo seleccionado por el parámetro *modo*. El resto de los parámetros que recibe son la ecuación del plano sobre el que se va a proyectar la sombra, y la posición de la luz. Es importante tener en cuenta que en caso de que el modo activo de sombreado sea `SOMBRAS_SUAVES`, la posición de la luz se verá modificada.

`void activar(GLenum modo, float groundPoints[3][3], float lightPos[4])`: Activa el sombreado con el modo seleccionado por el parámetro *modo*. El resto de los parámetros que recibe son tres puntos del plano sobre el que se va a proyectar la sombra y la posición de la luz. Es importante tener en cuenta que en caso de que el modo activo de sombreado sea `SOMBRAS_SUAVES`, la posición de la luz se verá modificada.

`void activaMarcadoStencil()`: Activa el marcado sobre el *buffer de estarcido*. Se debe llamar a esta función cuando se vaya a usar el *buffer de estarcido*, y se quieran marcar los píxeles sobre los que se quiere que se dibuje la sombra (o el reflejo). Después de esta llamada, deberán dibujarse las superficies sobre las que se quiere que se dibuje la sombra. Estas superficies no se dibujaran en la escena, pero el buffer quedará marcado.

`void activaComprobacionStencil(GLenum modo)`: Esta función desactiva el marcado sobre el *buffer de estarcido*, y activa la comprobación de los valores del buffer según el parámetro *modo*. Se debe llamar a esta función después de marcar el buffer (llamando a la función *activaMarcadoStencil* y dibujando las superficies receptoras de las sombras). Después de esto, se deberá llamar a la función *activar*, con cada uno de los planos de proyección. El valor del parámetro *modo* variará según el efecto deseado, pero será uno de los siguientes:

- `GL_KEEP`: Este modo se utilizará habitualmente en los reflejos, que se deberán hacer ANTES de las sombras, en caso de que se hagan ambos.
- `GL_INCR`: Este modo se utilizará habitualmente en las sombras, e impide el doble *blending*.

`void desactivaComprobacionStencil()`: Desactiva las comprobaciones sobre el *buffer de estarcido*.

`void desactivar()`: Desactiva las proyecciones de las sombras, eliminando las transformaciones realizadas por la llamada a la función *activar*.

`int getNumSombras()`: Devuelve el número de sombras a dibujar. El valor devuelto dependerá del tipo de sombras a dibujar. En el caso de que el modo `SOMBRAS_DURAS` esté seleccionado, el valor devuelto será 1. Si el *Sombreador* está configurado para mostrar `SOMBRAS_SUAVES`, entonces el valor devuelto será 3.

`void setNumSombras(int ns)`: Establece el número de sombras que compondrán la sombra suave. En caso de que el modo actual sea `SOMBRAS_DURAS`, o el parámetro *ns* sea menor o igual que 1, no se cambiará el valor actual.

`void setColorSombra(float r, float g, float b)`: Configura el color en el que las sombras serán mostradas. Los tres parámetros que reciben definen la cantidad de rojo (parámetro *r*), de verde (parámetro *g*) y de azul (parámetro *b*) del color de las sombras.

`bool getUsaStencil()`: Devuelve **true** si el uso del *buffer de estarcido* está activado, y **false** en caso contrario.

`void setUsaStencil(bool us)`: Configura el uso del *buffer de estarcido*. Si se establece el uso del buffer (valor **true** del parámetro *us*), entonces se realizará la comprobación sobre el buffer de estarcido al dibujar las sombras.

Formas de Uso y Ejemplos

Veamos en primer lugar cómo tendríamos que utilizar el objeto *Sombreador* para dibujar tanto sombras suaves como sombras duras. Lo primero que tenemos que haces es crear el objeto sombreador:

```
Sombreador* sb = new Sombreador();
// Sombreador* sb = new Sombreador (tipo)
```

En el caso de la segunda llamada (la que está entre comentarios), el valor de tipo tendría que ser alguno de los dados en la sección **Funciones**. Una vez creado el objeto, tendremos varias posibilidades. Para mostrar sombras utilizando el *buffer de estarcido*, tendremos en primer lugar que marcar el buffer. Tendremos, pues, el siguiente código:

```
DrawFloor (); // Dibujamos el suelo
sb->activaMarcadoStencil();
// Dibujamos el suelo sólo en el stencil buffer
DrawFloor();
sb->activaComprobacionStencil(GL_INCR);
```

Ahora, una vez marcado el buffer de estarcido, y activada la comprobación sobre el buffer, lo que tenemos que hacer es activar el sombreador, para que proyecte las sombras sobre el plano deseado (en este caso, el suelo). Para tener el mismo código para las sombras suaves y las duras, lo que haremos será un "**for**", con recorrido desde 0 hasta el número de sombras, ya que en el caso de las sombras duras este será 1. El código será como sigue:

```
for (int i = 0; i < sb->getNumSombras(); i++)
{
     sb->activar (groundplane, pos_luz);
     DrawObject();
     sb->desactivar();
}
DrawObject();
```

De esta forma, tendremos el mismo código para mostrar tanto sombras suaves como sombras duras. Vemos que después de dibujar la escena, dibujamos el objeto en sí.

El objeto sombreador también se puede utilizar para otros efectos, como podría ser el de los reflejos sobre el suelo (como ocurriría, por ejemplo, con un suelo de mármol, o sobre el agua de un lago). Para hacer esto, primero tendremos que construir un *Sombreador*. El modo en el que lo construyamos, realmente, es indiferente, ya que no vamos a mostrar sombras. Lo que querremos hacer es aprovecharnos de las funciones de activación y comprobación sobre el *buffer de estarcido*. Lo que tendremos que hacer, después de crear el *Sombreador*, será lo siguiente:

```
sb->activaMarcadoStencil();
// Dibujamos el suelo en el stencil buffer
DrawFloor();
sb->activaComprobacionStencil(GL_KEEP);
glPushMatrix();
// Invertimos la imagen, para dibujar el reflejo
glScalef(1.0, -1.0, 1.0);
DrawScene();
glPopMatrix();
// Activamos el blending con la función GL_ONE, GL_ONE
// para que se dibujen tanto el suelo como el reflejo.
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
// Dibujamos el suelo
DrawFloor ();
```

De esta forma, se obtiene el código necesario para implementar reflejos de forma sencilla. Si vamos a dibujar sombras posteriormente, tendremos que llamar a `activaComprobacionStencil(GL_INCR)` antes de dibujarlas, para evitar el doble blending.

Iluminación Estática

Técnicas para iluminar escenas sin utilizar la Iluminación de OpenGL

Introducción

Para conseguir mayor velocidad en las aplicaciones en tres dimensiones es muy usual recurrir a la técnica de "Pre-calcular todo lo posible". De esta forma si conocemos de antemano que luces van a influir en determinados polígonos, se puede calcular la iluminación una vez y mostrar los resultados en cada frame de la animación. Evitamos que OpenGL utilice la iluminación dinámica de esos polígonos en cada frame y así se gana mucho tiempo de ejecución. Naturalmente, para aplicar estas técnicas es necesario conocer la posición de las luces y de los polígonos y que su posición relativa (luces-polígonos) no cambie durante la ejecución del programa.

Pre-Iluminacion

La primera técnica consiste en determinar el color que tendrá un vértice del polígono que se quiere iluminar. Primero se definen las posiciones de las luces estáticas y sus propiedades. Después se itera en cada vértice del polígono calculando el color resultante bajo el efecto de las luces estáticas. Para cada vértice también se debe especificar las propiedades del material, tal y como se hace también en OpenGL. Una vez almacenado el color resultante de cada vértice, se renderiza el polígono sin activar la iluminación de OpenGL y se utiliza el método *GORAUD* para rellenar el color del polígono. Este método interpola el color de los vértices para decidir el color de los pixels del interior del polígono.

La pre-iluminación da el mismo resultado visual que si se utilizara la iluminación dinámica de OpenGL. Si no, sería inservible.

Otra ventaja resultante del precálculo es que ya no se tiene límite en el número de luces que se pueden activar. En OpenGL existen un máximo de 8 luces.

Ventajas

- Velocidad de ejecución.
- Número de luces ilimitado.
- Misma apariencia que la iluminación dinámica.

Desventajas

- La posición relativa luces-polígonos no debe cambiar.
- Requiere una malla de polígonos pequeños para una iluminación más realista (también se requiere en iluminación dinámica).
- No puede tratar la componente especular.

Encapsulación de las luces y del material de OpenGL

Siguiendo la tradición del proyecto, se ha encapsulado la funcionalidad de la iluminación de OpenGL en una clase. Se ha creado la clase abstracta `CLuz` que contiene los métodos y propiedades que cualquier punto de luz debería tener. De esa clase abstracta se han derivado otras dos, `CluzDinamica` y `CluzEstatica`. La primera se puede utilizar como enfoque OO de la iluminacion de OpenGL, con la misma filosofía que la clase `CTextura`. La segunda se utiliza para cualquier cálculo de iluminación que requiera de los parámetros de una luz.

La clase CluzDinámica facilita la iluminación de OpenGL ya que evita al programador tener que preocuparse por el identificador de la luz. Internamente, escoge el primer identificador libre y no deja crear más luces que el máximo establecido por OpenGL.

El Material del vértice también se ha encapsulado en la clase `Material`. Así el color resultante de un vértice está en función de las luces que influyen en el vértice y el material del vértice.

La clase `CLuzEstatica` contiene los métodos `calculaColor*(...)` que dados un vértice y un material, calculan las componentes de color generadas por esa luz. Internamente usan las mismas ecuaciones de iluminación que OpenGL.

Nota: No se puede calcular la componente especular porque depende de la posición del observador y por tanto es de naturaleza dinámica. Los polígonos iluminados estáticamente no pueden tener reflejos.

Función de la clase CLuz	Significado
`setPos(float` *`px`*`,float` *`py`*`,float` *`pz`*`,float` *`pw`*`)`	Fija la posición de la luz en (*px*, *py*, *pz*) si *pw* = 0. Si *pw* = 1, la luz irradia en la dirección dada pero no esta posicionada en ningún lugar específico.
`setAmbiente(float` *`ar`*`,float` *`ag`*`,float` *`ab`*`,float` *`aa`*`)`	Fija la componente ambiente de la luz.
`setDifusa(float` *`dr`*`,float` *`dg`*`,float` *`db`*`,float` *`da`*`)`	Fija la componente difusa de la luz
`setEspecular(float` *`sr`*`,float` *`sg`*`,float` *`sb`*`,float` *`sa`*`)`	Fija la componente especular de la luz
`setDireccion(float` *`sx`*`, float` *`sy`*`, float` *`sz`*`)`	Fija la dirección de la luz.
`setExpSpotLight(float` *`exp`*`)`	Fija la potencia de la luz.
`setCutOffSpotLight(float` *`spot_cutoff`*`)`	Fija el ángulo de abertura de la luz

`setAtenuacion(float atenuacion)`	Fija el coeficiente constante de atenuación.
`setAtenuacionLineal(float atenuacion_lineal)`	Fija el coeficiente lineal de atenuación
`setAtenuacionQuad(float atenuacion_quad)`	Fija el coeficiente cuadrático de atenuación.
`void render()`	Renderiza un punto de luz en la posición actual con el color de la componente especular.
`void activa()`	Activa la luz para ser utilizada.
`void desactiva()`	Desactiva la luz

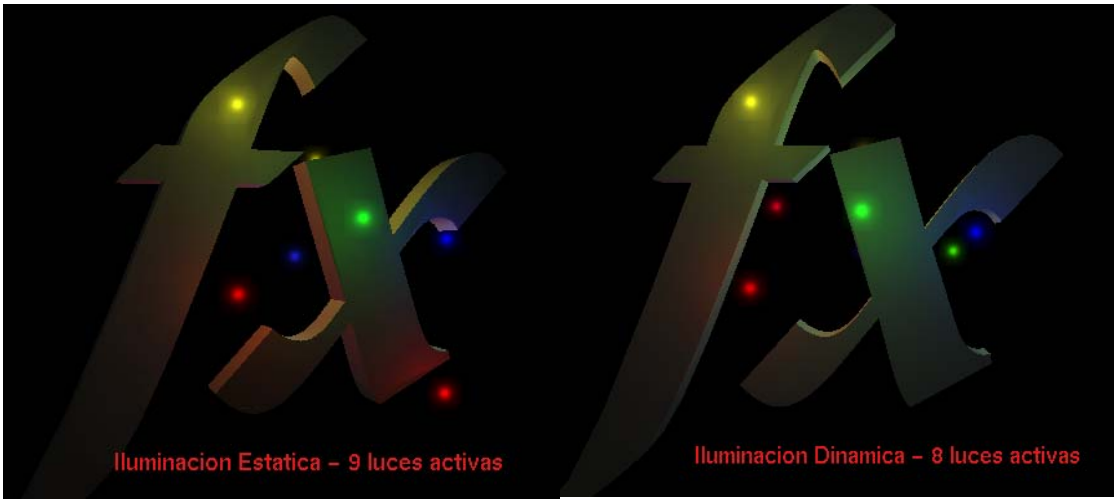
Función de la clase CLuzDinamica	**Significado**
`void activa()`	Activa la luz para ser utilizada en OpenGL
`void desactiva()`	Desactiva la luz de OpenGL

Función de la clase CLuzEstatica	**Significado**
`void activa()`	No realiza nada ya que es una luz estática.
`void desactiva()`	No realiza nada ya que es una luz estática.
`calculaColor(float x, float y, float z, float nx, float ny, float nz, Material* mat, float *rojo, float *verde, float *azul)`	Añade al color (*rojo*, *verde*, *azul*) el color resultante en el vértice (*x*, *y*, *z*) con el material *mat* y normal (*nx*, *ny*, *nz*) producido por la luz estática desde la que se invoca el método.
`calculaColorfv(float *p, float* n, Material* mat, unsigned char *color)`	Misma función que la anterior pero refiriéndose a los elementos mediante punteros.

Función de la clase Material	**Significado**
`void activar()`	Activa el material para ser aplicado en OpenGL a la geometría especificada
`setAmbiente(float r,float g,float b,float a)`	Fija la componente ambiente del material.
`setDifusa(float r,float g,float b,float a)`	Fija la componente difusa del material
`setEspecular(float r,float g,float b,float a)`	Fija la componente especular del material
`setIndiceReflexion(float i)`	Fija el índice de reflexión del material.
`setEmision(float r,float g,float b,float a)`	Fija el color que emite el material.

Se incluye también la función `calculaColores(...)`, a modo de demostración, utilizada para calcular la iluminación de un modelo entero. Se especifica el array de vértices, caras, normales, luces y material y devuelve un array con los colores finales de cada vértice.

Función	Significado
`void calculaColores(float *vertices, float *normales, unsigned int *caras, unsigned int numCaras, Material* mat, LuzEstatica** luz,unsigned int numLuces, unsigned char** colores)`	Calcula el color resultante de la malla especificada por (*vertices*, *normales*, *caras*) con *numCaras* caras producido por el *mat* aplicado a la malla y por la lista de luces *luz*. El resultado se almacena en el array *colores*, con la misma longitud que la lista de vértices.



LightMapping

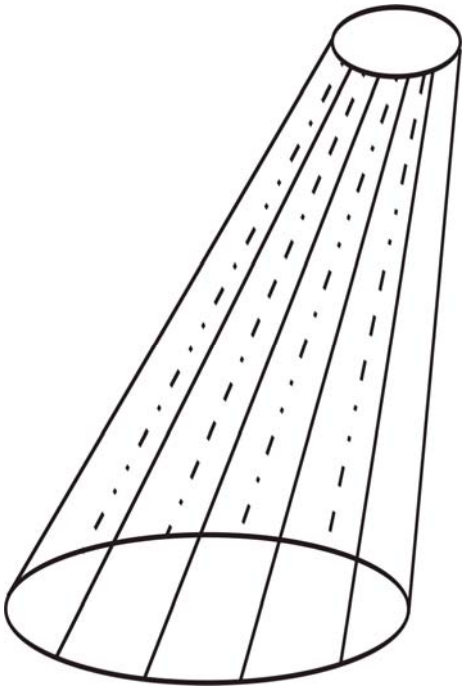
Esta técnica consiste en utilizar texturas para la iluminación de los polígonos. Requiere también de antemano conocer las posiciones de las luces con respecto a los polígonos. Se utiliza para la iluminación de una imagen en escala de grises que indica como se ilumina cada parte de la textura. La principal ventaja de esta técnica es que puede iluminar de forma compleja un solo polígono. Si hubiera que hacerlo con iluminación tradicional (por vértice) habría que dividir el polígono en muchos otros más pequeños. Así hay más vértices y la iluminación se realiza de forma más cercana a la iluminación por píxel.

Requiere que el hardware soporte multitexturado. En la explicación del objeto `CTextura` se presenta el código de cómo realizar *lightmapping*, así que en esta sección lo obviaremos.

Volúmenes de Luz

Cuando la luz incide en una atmósfera con partículas en suspensión, éstas reflejan la luz y se produce el efecto del volumen de luz. El volumen de luz es el espacio comprendido entre el punto de luz y la zona iluminada, describe las trayectorias que seguirían los fotones. Esta técnica, al contrario que las anteriores no es totalmente estática. Se puede calcular el volumen de luz en tiempo de ejecución sin ningún impacto importante en el tiempo de ejecución.

El volumen de luz consiste normalmente en un cono. La parte superior corresponde al punto de luz y la inferior a la superficie iluminada. Se trata de dibujar este cono con un efecto de *blending* para que ilumine los pixels que ya están en el framebuffer. Esta técnica requiere por tanto que los volúmenes de luz sean dibujados al final del renderizado de la escena.



Para evitar que se dibuje la parte del cono que no encara al espectador, se activa la eliminación de caras que no encaren al observador con el método `glEnable(GL_CULL_FACE)`. Al dibujar el cono de luz, se debe activar la mezcla para que sume luz a los píxeles que ya estén en el framebuffer. Esto implica dos cosas. Primero que el cono de luz debe ser dibujado al final y segundo que se deben eliminar las caras traseras para que no hayan zonas donde se ilumine dos veces.

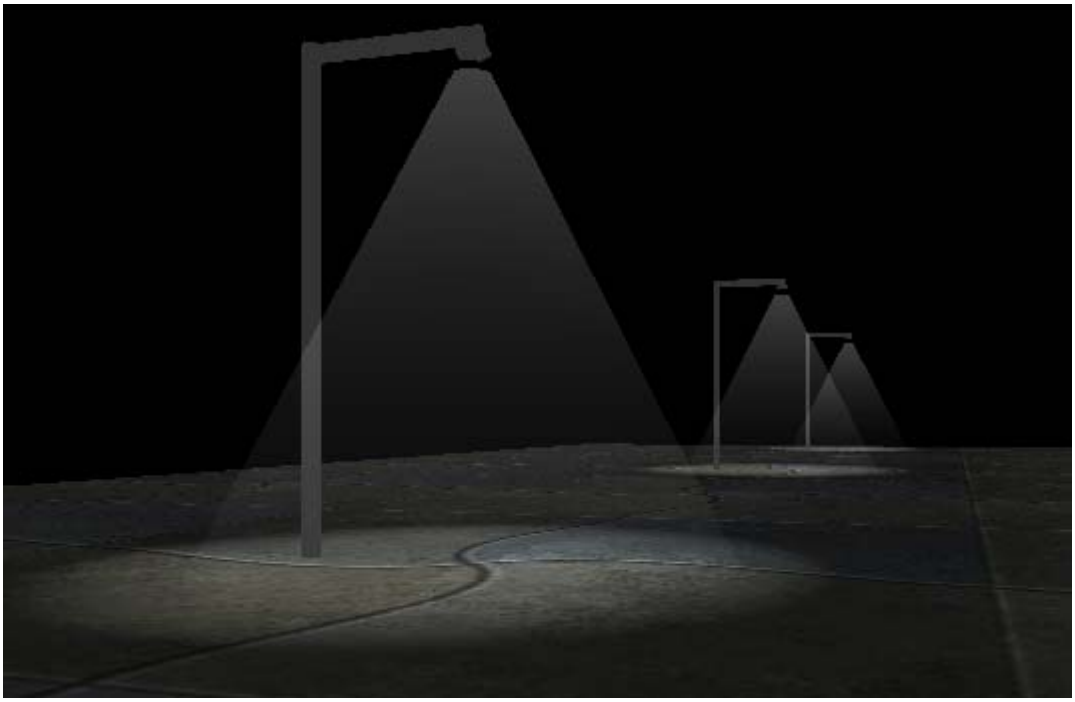
Para mejorar el efecto, el alpha de la zona superior del cono es mayor que la inferior y así se consigue mayor intensidad de luz cerca del foco.

Ejemplos

El ejemplo staticlight.bpr incluido en el CD demuestra la técnica de la iluminación estática. Se puede medir el impacto en el tiempo de ejecución comparando la ejecución con el del proyecto dinamiclight.bpr. Esta diferencia se agranda en sistemas

sin acelerado de gráficos. En la iluminación estática se utilizan 9 luces mientras que en la iluminación dinámica se utilizan 8 (el máximo posible).

El ejemplo lightmap.bpr muestra como utilizar los *lightmaps* para realizar una iluminación con esta técnica. También utiliza volúmenes de luz para hacer la presentación más realista.



Renderizado de líneas con patrones avanzados

Técnica para renderizar líneas con texturas

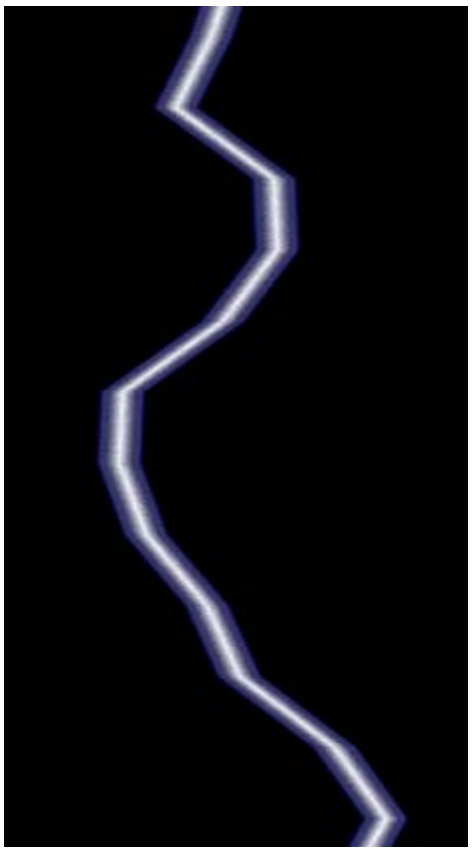
Introducción

El objetivo de este apartado es conseguir renderizar líneas que tengan una apariencia más gruesa. Ejemplos de estos tipos de línea pueden ser luces de neón, rayos láser, relámpagos…

Renderizado de QUADS_STRIP y LINEAS

La forma de conseguirlo es envolver a la línea con grosor para poder aplicarle una textura. Resulta muy adecuado utilizar QUAD_STRIP, ya que por cada punto de la línea sólo hay que generar dos vértices, al primero se le resta una constante, y al segundo se le suma. Conjuntamente utilizado con *billboarding*, hemos conseguido dotar de grosor a la línea. Sólo resta aplicar la textura y finalizado. Como se puede observar, es una técnica muy sencilla pero con resultados fantásticos.

Ejemplo: Rayo



Para ilustrar esta técnica nos hemos decidido por el rayo de una tormenta porque presenta además la dificultad añadida de generar la trayectoria del rayo.

El rayo está dividido en segmentos de longitud fija, y con una función aleatoria uniforme se decide el desplazamiento que tendrá el siguiente segmento.

Una vez generada la línea, se recubre con QUADS y se aplica la textura. En este caso se utilizó multitextura. Una textura para el color que es simplemente una degradación azul-blanco-azul y otra que genera un poco de distorsión en el dibujo con la misma técnica que el *lightmapping*.

La demostración se puede hallar en el CD en el proyecto de C++ Builder rayo.bpr.

Simulación de superficies ondulantes

Introducción

Las superficies que pretendemos simular en este apartado tienen una amplia gama de aplicaciones a la hora de dar realismo a una escena. Podemos simular banderas ondeadas al viento, superficies líquidas o viscosas, plásticos, etc. Las características físicas de estos objetos son muy diversas, y debido a la imposibilidad de modelar todas con exactitud (sobre todo en una aplicación interactiva o de tiempo real) se deben elegir modelos que proporcionen sencillez, realismo y eficiencia.

Tipos de superficies

Las superficies ondulantes son objetos bidimensionales que pueden ser recorridos por ondas de dos tipos: unidimensionales o bidimensionales. Elegimos, por la naturaleza totalmente distinta de ambas, modelar los dos tipos en sendas clases.

WavingObject: Representa una superficie recorrida por una onda unidireccional continua y repetitiva.

TwoDimensionalWavingObject: Representa una superficie recorrida por una onda bidimensional que se extiende a lo largo y ancho de aquella. Para proporcionar mayor realismo, elegiremos un modelo matemático que permita que los frentes de ondas interactúen entre ellos, creando patrones de interferencia y posibilitando una interacción dinámica con el objeto.

Modelo matemático

En cuanto a las ondas unidimensionales, el modelo es muy simple. Se basa en la ecuación de onda unidimensional:

$$z = A\cos(\frac{2\pi}{\lambda}x - \frac{2\pi\nu}{T}t)$$

Éste será la ecuación que usemos para hallar el desplazamiento inicial y en cada paso de la onda en z.

Para darle mayores posibilidades, añadimos un factor de atenuación, que disminuye o aumenta la amplitud de la onda a medida que ésta se desplaza por la superficie:

$$z = A\cos(\frac{2\pi}{\lambda}x - \frac{2\pi\nu}{T}t) \quad *\text{Att(x)}$$

La función de atenuación Att(x) es una función lineal que varía en función de la posición, alcanzando su punto máximo en un extremo (Att(x)= 1)y el mínimo en el otro (Att(x) = at.inicial).

Como puede observarse, la implementación de este modelo es muy sencilla y razonablemente eficiente. Se podría mejorar esta eficiencia (cuyo principal enemigo es la función trigonométrica que debemos calcular) de dos formas:

- Tabulando los posibles valores de la función coseno de tal modo que su calculo se reduzca a una búsqueda en una tabla hash, idealmente en tiempo constante.
- Calculando los valores de las posiciones a partir de los valores anteriores. Esto hace el código más eficiente pero menos legible y hace que efectos como la atenuación sean más complicados de implementar.

En cuanto al modelo de onda bidimensional, resulta ligeramente más complejo.
Partimos de la ecuación de una onda bidimensional:

$$z = c^2 \frac{\partial^2 x}{\partial\, t^2}\frac{\partial^2 y}{\partial\, t^2}$$

Resolviendo esta ecuación diferencial para una rejilla cuadrada discreta con número de secciones L tenemos:

$$z(x,y,t) = \frac{2}{L}\sum_{n}^{\infty}\sum_{m}^{\infty} A_{mn}\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L})\cos(c\omega t)$$

$$\omega = \frac{\pi}{L}\sqrt{(mx)^2 + (ny)^2}$$

Los coeficientes Amn se hallan evaluando las integrales:

$$A_{mn} = \frac{2}{L}\int_0^L\int_0^L f(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L})dxdy$$

donde f(x,y) es la forma inicial del agua. Estas ecuaciones resultan difíciles de resolver eficientemente. Pero al discretizar el espacio en una malla, tenemos que podemos resolverlas mediante una transformada de Fourier. Aunque esta solución sería demasiado costosa en tiempo todavía. Por ello buscamos un método más eficiente, como es la aproximación de las derivadas parciales mediante *diferencias centrales*:

$$\frac{z_{i,j}^{n+1} - 2z_{i,j}^{n} + z_{i,j}^{n-1}}{\Delta t^2} = c^2\left(\frac{z_{i+1,j}^{n} + z_{i-1,j}^{n} + z_{i,j+1}^{n} + z_{i,j-1}^{n} - 4z_{i,j}^{n}}{h^2}\right)$$

Siendo los z las sucesivas alturas con coordenada espacial los subíndices y temporal los superíndices, y h el lado de cada sección de la malla.
Tenemos entonces que:

$$z_{i,j}^{n} = \frac{c^2\Delta t^2}{h^2}(z_{i+1,j}^{n} + z_{i-1,j}^{n} + z_{i,j+1}^{n} + z_{i,j-1}^{n}) + (2 - \frac{4c^2\Delta t^2}{h^2})z_{i,j}^{n} - z_{i,j}^{n-1}$$

Como se puede observar el movimiento en un punto en concreto sólo se ve influenciado por sus 4 vecinos más próximos. Como h es constante, $1/h^2$ puede ser precalculado. Con lo cual obtenemos un código sumamente eficiente tanto en tiempo, al estar compuesto de sumas y multiplicaciones únicamente, como en espacio, ya que podemos guardar todos los valores en un único array bidimensional que represente la superficie. Este proceso numérico se vuelve inestable cuando:

$$\frac{c^2\Delta t^2}{h^2} > \frac{1}{2}$$

Para evitar esto, fijamos $h^2 = 2c^2\Delta t^2$. De este modo el tamaño de las secciones determinará la velocidad de la onda. Con este modelo logramos simular una onda interactiva consigo misma, cuyos frentes de ondas producen interferencias y rebotan en los bordes de la superficie.

Guía de uso

Los interfaces de los objetos son los siguientes:

WavingObject:

Función	Significado
`void draw()`	Dibuja la superficie a lo largo del plano x-y y centrada en el punto (0,0).
`void loadTexture(CTextura* t)`	Asigna una textura a la superficie.
`void setColor(GLfloat r,GLfloat g,GLfloat b,GLfloat a)`	Asigna un color a la superficie.
`void setInitialAmplitude(GLfloat z)`	Fija la amplitud inicial, es decir, el desplazamiento en z máximo de la onda.
`void setSections(int x)`	Fija el número de secciones en las que estará dividida la superficie.

`void setSize(GLfloat x, GLfloat y)`	Fija el tamaño de la superficie.
`void setWavelength(GLfloat l)`	Fija la longitud de onda relativa a la superficie: cuando es 1 la onda entera recorre la superficie exactamente una vez.
`void setAttenuation(GLfloat a)`	Fija la atenuación máxima que tendrá la onda en el punto extremo: 0 significa sin atenuación, 1 implica que el extremo no se moverá. Esto puede ser útil por ejemplo para simular banderas atadas a un mástil por ese extremo.
`void setSpeed(int speed)`	Fija la velocidad de la onda relativa al número de segmentos de la malla.
`bool createObject()`	Crea el objeto una vez se le han asignado todos los parámetros anteriores.
`void step()`	Avanza la onda en un paso.

TwoDimensionalWavingObject:

Función	**Significado**
`void setSize(GLfloat x)`	Fija el tamaño de la superficie. Como se puede ver esta será cuadrada.
`void setSegments(int x)`	Fija el número de segmentos en los que estará dividida la onda.
`void setInitialAmplitude(GLfloat a)`	Fija la amplitud de la onda en el punto y momento inicial.
`void setInitialPoint(int x, int y)`	Fija el punto inicial donde comenzará el movimiento en el sistema de coordenadas de la malla: Cada unidad representa una sección, y el origen se halla en la esquina inferior izquierda.
`void setTexture(CTextura *t)`	Asigna una textura a la superficie.
`void createObject()`	Crea el objeto una vez se le ha asignado valores a todos sus parámetros.
`void draw()`	Dibuja la superficie a lo largo del plano x-y y centrada en el punto (0,0).
`void step()`	Avanza la onda un paso.

Debemos tener en cuenta que para simular correctamente las propiedades de fluidos este modelo resulta a veces demasiado sencillo. Se debe modificar los parámetros tanto de amplitud inicial como de tamaño de sección (que determina la velocidad) para darle un aspecto característico.

Ejemplo

En el ejemplo podemos observar que el rendimiento no cae debido a que no utilizamos funciones costosas. También podemos experimentar para darle a la superficie que representa el agua otros aspectos.

Conclusiones

La naturaleza del proyecto nos ha obligado a investigar en muchos frentes. El primero ha sido lógicamente OpenGL. Hemos conseguido dominar la librería gráfica en profundidad, eficiencia incluida. La eficiencia ha sido el pilar central junto con la facilidad de uso de las clases desarrolladas. Las extensiones de OpenGL nos han permitido conocer el hardware gráfico para poder sacar partido de él. Precisamente la potencia del hardware gráfico ha sido la gran olvidada de Ingeniería Informática, y ese vacío es el que hemos tratado de llenar.

Por otro lado, hemos consultado principalmente con los fabricantes de hardware gráfico, con documentos de programadores gráficos, técnicas de diseñadores de gráficos... para poder establecer esa unión tan necesaria (principalmente en los videojuegos) entre programadores y diseñadores. Pensamos que las clases desarrolladas sirven tanto al programador experto de OpenGL como al diseñador profano en esta materia, poniendo el punto de atención en los efectos gráficos y no en el lenguaje de programación. De esta forma, el usuario de la librería se centra en lo que quiere hacer y no en como lo tiene que hacer. Obviamente esto implica un proceso de desarrollo de la aplicación más rápido.

Cabe destacar la variedad de técnicas tratadas en el proyecto. Obviamente quedan otras muchas en el tintero, pero con la base que proporciona cada apartado, el lector puede avanzar con sencillez. Estas otras técnicas pueden ser el *motion blur*, *bump mapping*, *motif lighting*...

El grupo ha quedado bastante satisfecho con la labor realizada, especialmente en el apartado del diseño, en el que se han invertido muchas horas para decidir que interfaz podía ser más agradable al cliente. También mencionar las demostraciones proporcionadas en el CD. No nos hemos quedado en la implementación, sino que hemos querido mostrar la potencia de estas técnicas, que aun siendo simples, proporcionan resultados atractivos.

Por último destacar que el cliente puede utilizar las clases sin miedos a ineficiencias ya que utilizan la capacidad del hardware gráfico siempre que esté disponible.

Agradeceríamos cualquier comentario o sugerencia.

Bibliografía

- Game Programming Gems editado por Mark DeLoura.
- Game Programming Gems 2 editado por Mark DeLoura.
- OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1,2 por Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, OpenGL Architecture Review OpenGL Game Programming
- Computer Graphics: Principles and Practice por James D. Foley, Andries van Dam, Steven K.Feiner, John F Hughes
- Graphics Gems por Andrew S. Glassner
- Mathematics for 3D Game Programming & Computer Graphics por Eric Lengyel
- OpenGL Game Programming por DaveAstle, Kevin Hawkins, Andre LaMothe
- NVidia Developer Home developer.nvidia.com
- ATI developer www.ati.com/developer/
- NeHe Productions nehe.gamedev.net/
- Game Tutorials gametutorials.com/Tutorials/OpenGL